Secure Lifecycle Management for Internet of Things Devices

Tolgahan Akgün

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo, Finland 31.7.2020

Supervisors

Prof. Tuomas Aura, Aalto University
Prof. Danilo Gligoroski, NTNU

Advisors

Sandeep Tamrakar, D.Sc. (Tech.), Huawei Technologies
Philip Ginzboorg, D.Sc. (Tech.), Huawei Technologies

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Tolgahan Akgün

Title Secure Lifecycle Management for Internet of Things Devices

Degree programme Master's Programme in Security and Cloud Computing (SECCLO)

Major Security and Cloud Computing **Code of major** SCI3084

Supervisors Prof. Tuomas Aura, Aalto University
Prof. Danilo Gligoroski, NTNU

Advisors Sandeep Tamrakar, D.Sc. (Tech.), Huawei Technologies
Philip Ginzboorg, D.Sc. (Tech.), Huawei Technologies

Date 31.7.2020 **Number of pages** 54 **Language** English

Abstract

In recent years, IoT devices have been adopted for various uses cases including for home applications such as smart lighting and heating and cooling systems. The IoT devices are simple and constrained devices. Usually, these simple devices are paired with and managed by controller devices such as smartphones over home wireless network. The pairing protocol along with the command and control protocols between the IoT device and the smartphone are usually proprietary. Therefore, users are required to install a dedicated application to access and control each brand and type of device. LwM2M has been designed as an open standard to increase interoperability between the simple devices from different ecosystems. It can be used to secure the connection between the simple device and the controller. The LwM2M protocol uses pre-shared keys, raw public keys, and X.509 certificates for authentication. However, these authentication methods have some deployment and scalability problems, and out-of-band authentication has been suggested as an alternative. This thesis project aims to adapt the LwM2M protocol for secure device pairing and lifecycle management for Internet of Things device in such a way that it can be used with out-of-band authentication. A proof-of-concept prototype has been implemented with Raspberry Pi 3 B+ as the simple device and an Android smartphone as the controller.

Keywords IoT security, Out-of-band, Authentication, QR code, OMA LwM2M

Preface

I want to thank Professor Tuomas Aura, Professor Danilo Gligoroski and my instructors Sandeep Tamrakar and Philip Ginzboorg.

Otaniemi, 31.7.2020

Tolgahan Akgün

Contents

Abbreviations and Acronyms

AES	Advanced Encryption Standard
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CoAP	Constrained Application Protocol
CRUD	Create, Read, Update and Delete
DDoS	Distributed Denial of Service
DER	Distinguished Encoding Rules
DoS	Denial of Service
DSA	Digital Signature Algorithm
DTLS	Datagram Transport Layer Security
DVR	Digital Video Recorder
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDHE	Elliptic Curve Diffie-Hellman Ephemeral
ECDSA	Elliptic Curve Digital Signature Algorithm
EdDSA	Edwards-Curve Digital Signature Algorithm
FreeRTOS	Free Real-time Operating System
GPIO	General Purpose Input/Output
HAP	HomeKit Accessory Protocol
HTTP	Hypertext Transfer Protocol
ID	Identity
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IoT	Internet of Things
IP	Internet Protocol
IPSec	Internet Protocol Security
LED	Light Emitting Diode
LoRaWAN	Long Range Wide Area Network
LwM2M	Lightweight Machine-to-Machine
M2M	Machine-to-Machine
mDNS	Multicast Domain Name System
MFi	Made for iPhone/iPod/iPad
MitM	Man-in-the-middle
NAT	Network Address Translation
NFC	Near Field Communication
NIST	National Institute of Standards and Technology
OMA	Open Mobile Alliance
OOB	Out-of-band
OS	Operating System
PEM	Privacy Enhanced Mail
PKI	Public Key Infrastructure
PoC	Proof-of-Concept
PSK	Pre-shared Key

PWM	Pulse Width Modulation
QR Code	Quick Response Code
RAM	Random Access Memory
REST	Representational State Transfer
RFC	Request for Comments
RFID	Radio-frequency Identification
ROM	Read-only Memory
RPK	Raw Public Key
RSA	Rivest-Shamir-Adleman (cryptosystem)
SD Card	Secure Digital Card
SHA	Secure Hash Algorithm
SMS	Short Messaging Service
SoC	System on a Chip
SRP	Secure Remote Password
SSH	Secure Shell
SSID	Service Set Identifier
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
USB	Universal Serial Bus
VPN	Virtual Private Network
WiFi	Wireless Fidelity
WLAN	Wireless Local Area Network
WPA2	Wi-Fi Protected Access 2

1 Introduction

The Internet of Things (IoT) is a network of devices that can communicate without the need of a human intervention. The IoT term was first introduced by Kevin Ashton in 1999 [1]. In the early stages, the use of IoT was limited to Radio-Frequency Identification (RFID) to manage supply chains. Later, the IoT concept evolved to include other kinds of connected devices like wearables, home appliances, various sensors and lighting.

There is a high variation in the IoT devices capabilities, technologies, and quality of implementation, which means that there is also a high variation in their security level. Also, the number of IoT devices has been increasing rapidly, and this increase is predicted to continue. According to a survey by Ericsson, the total number of IoT devices is expected to increase on the average at the rate of 15 % annually [2] between 2019 and 2025, from 10.8 billion to 24.9 billion devices.

The growing use of IoT devices worldwide increases the risks of breaches of privacy and data security. This, in turn, implies that there is higher potential for attacks involving IoT devices [3]. The threat is that systems containing IoT devices may be vulnerable to attacks via those devices when the connected "things" are exploited by attackers. As one example, in August 2016, the Mirai botnet consisting of webcams, Digital Video Recorders (DVRs) and routers was used to launch a Distributed Denial of Service (DDoS) attacks on popular web sites including Twitter, Netflix, Reddit, and GitHub [4]. According to a survey by F-Secure, the number of attack events measured from January through June 2019 was twelve times higher when compared with the same period in 2018, and they claimed that this increase was largely driven by IoT-related traffic [5].

IoT devices can be classified based on their capabilities. For example, in RFC 7228 [6] they are categorized based on the amount of random access memory (RAM) and code size, as shown in Table 2.

Classification	**Data Size (e.g., RAM)**	**Code Size (e.g., Flash)**
Class 0, C0	<<10 kB	<<100 kB
Class 1, C1	~10 kB	~100 kB
Class 2, C2	~50 kB	~250 kB

Table 2: Classification of IoT devices based on memory size [6].

Let us consider the impact of the memory size on the IoT device's connectivity. Devices in Class 0 (less than 10 kB of RAM and less then 100 kB of Flash) have severe constraints for the ability to communicate securely. These devices are typically are pre-configured and connected to proxies or gateways rather than directly to the internet [6].

It is challenging to employ a full protocol stack that includes protocols like Hypertext Transfer Protocol (HTTP) and Transport Layer Security (TLS) on the top of Trans-

mission Control Protocol (TCP) in devices of Class 1 (about 10 kB of RAM and about 100 kB of Flash) due to the memory and RAM space. However, those devices can use more lightweight protocols like Constrained Application Protocol (CoAP) and Datagram Transport Layer Security (DTLS) on the top of User Datagram Protocol (UDP). The code base and and processing requirements of the software have to be quite constrained [6].

Devices in Class 2 (about 50 kB of RAM and about 250 kB of Flash) can perform on a par with mobiles phones or notebooks in supporting most security protocols. However, the protocol implementations have to be lightweight and energy-efficient. Class 2 devices can be used to control and provide connectivity to the less capable Class 0 and 1 devices [6].

Device lifecycle management is a set of integrated steps which defines the entire lifecycle of a product including design, manufacturing, deploying, updating and disposing [7]. The management of IoT devices during their lifecycle is one of the key issues in the IoT network. One way to solve this issue is to design a custom-made set of protocols. This approach enables making a coherent, "turn-key" solution that fits well with the designer's needs. However, it leads to a closed, tightly-controlled IoT ecosystem. Apple HomeKit Accessory Protocol (HAP) [8] is an example of this approach. Section 2.2.1 describes HAP in more details.

Another way of solving this issue is adapting a set of standard protocols while aiming to change them as little as possible. The potential advantage of this approach is that it saves design and implementation time and enables an open IoT ecosystem. Specifically, this work has adapted the Lightweight Machine to Machine (LwM2M) specifications by Open Mobile Alliance [9] for the management of IoT devices in the consumer's home. In this case, the IoT device is part of the home Wireless Fidelity (WiFi) network and it is controlled via the smartphone of the user. (See Figure 1).

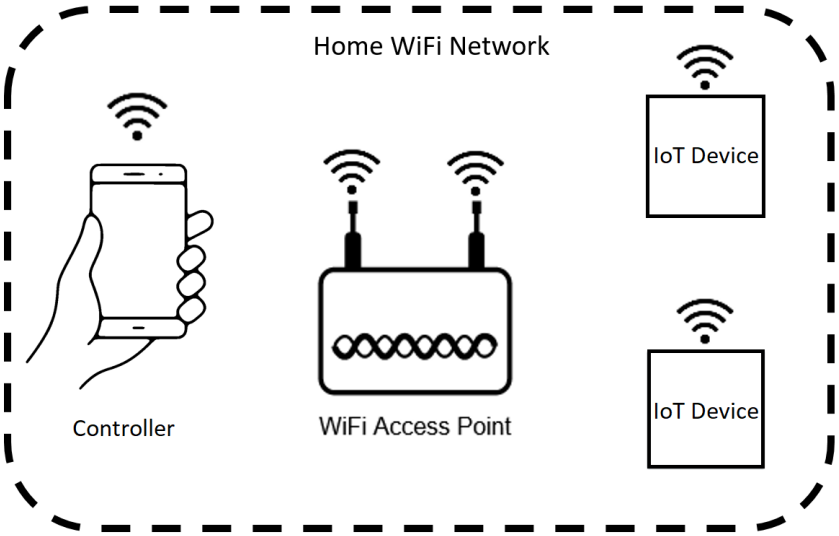

Figure 1: IoT devices in home WiFi network.

The LwM2M specifies an architecture, illustrated in Figure 2, which includes a client, IoT device, and two servers: LwM2M Bootstrap Server for setting up the client, and LwM2M Server for controlling and communicating with the client after the initial setup.

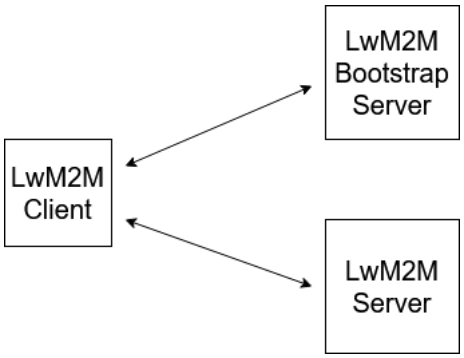

Figure 2: LwM2M entities.

LwM2M also defines an extensive library of application-level abstractions, called Objects, for managing and controlling resources in IoT devices using the Constrained Application Protocol (CoAP). CoAP has been designed for constrained devices by Internet Engineering Task Force (IETF) [10]. This thesis conceives the configuration shown in Figure 3, where CoAP runs on top of DTLS and UDP in the user's home WiFi network.

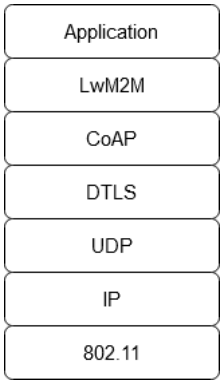

Figure 3: The protocol stack in our proof-of-concept implementation.

Our adaptations of LwM2M include (i) porting of the LwM2M Servers to a smartphone, and (ii) changing the initial setup of the IoT device (*bootstrap* in LwM2M terminology) so that it fits to the home use case.

A proof-of-concept prototype has been implemented on Raspberry Pi as the LwM2M Client and Android smartphone as the LwM2M Bootstrap Server and LwM2M Server to demonstrate the proposed approach. This thesis focuses on the lifecycle management of IoT device. The implementation of the LwM2M Servers in the Android smartphone is a subject of another master's thesis that was written in the same project [11].

The rest of this thesis is organized as follows. Section II describes the background and existing solutions related to this work. Section III presents the system description of our LwM2M adaptation and an overview of the proof-of-concept prototype implementation. Section IV contains additional details of the implementation. Section V discusses several aspects of the proposed system based on the findings during its design and implementation.

2 Background

2.1 Internet Standards

2.1.1 Constrained Application Protocol

Constrained Application Protocol (CoAP) is an application layer protocol designed for constrained devices. It was introduced in RFC 7252 [10] by IETF in 2014. The protocol provides a RESTful interface for constrained devices. It adopts HTTP-like Representational State Transfer (REST) design with a compact binary representation, which is easy to parse for constrained devices. It supports the create, read, update, delete (CRUD) operations like HTTP [10]. Therefore, CoAP messages can easily be translated into HTTP requests, which makes it easier to integrate with web applications.

The protocol targets simple devices, and it is possible to implement it with 8.5 kB of memory and 1.5 kB of RAM [12]. One CoAP message can be as small as 4 bytes. Thus, the simple design and simple message structure provide low bandwidth usage and low implementation complexity. CoAP carries messages between constrained devices and machine-to-machine (M2M) applications over lossy networks. HTTP cannot work efficiently on the datagram protocols [13]; however, CoAP can work efficiently on top of several transport protocols like UDP and Short Message Service (SMS) (it also supports other protocols, see Figure 4). The support for different datagram protocols makes it possible to use CoAP for one-to-many and many-to-one communication. LwM2M employs CoAP to transmit messages between LwM2M nodes. CoAP itself does not provide security. Instead, it uses DTLS or lower layer security protocols, such as Virtual Private Network (VPN) and Internet Protocol Security (IPSec), to secure the communication.

2.1.2 Lightweight Machine-to-Machine

The Lightweight machine-to-machine (LwM2M) protocol was introduced in 2017 by Open Mobile Alliance (OMA). It was designed to suit the needs of constrained IoT devices. Management and control of a device are integrated into one protocol. It supports remote control, firmware upgrade, certificate provisioning, access control policy and other functionality that is needed to manage an IoT device throughout its lifecycle.

The LwM2M protocol can run on top of different communication technologies like TCP, UDP, and Non-IP protocols like Low Power Wide Area Network (LoRaWAN). The support of different transport protocols makes LwM2M flexible and inter-operable with other systems. Figure 4 shows the supported protocols in different layers of the protocol stack.

LwM2M employs CoAP for exchanging data between the IoT devices, LwM2M Server and LwM2M Bootstrap Server [9]. CoAP can be regarded as a binary and lightweight version of HTTP, which is designed for constrained devices [14]. The CoAP protocol was designed specifically for IoT devices [10], since HTTP is not suitable due to the

long messages and complex parsing logic. A CoAP message can be as small as 4 bytes (details on CoAP in Section 2.1.1). CoAP itself does not provide message security; however, the CoAP messages can be protected using transport layer security such as TLS or DTLS or an application layer security such as Object Security for Constrained RESTful Environments (OSCORE) protocol [10, 15].

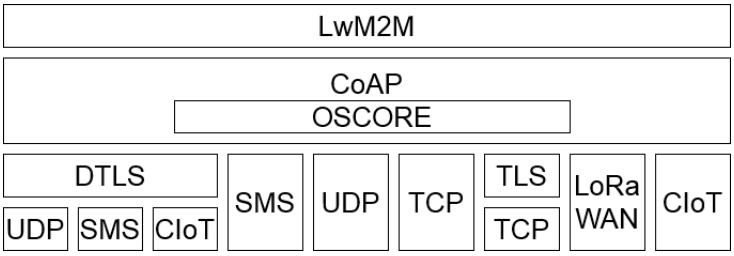

Figure 4: LwM2M protocol stack, adapted from [9].

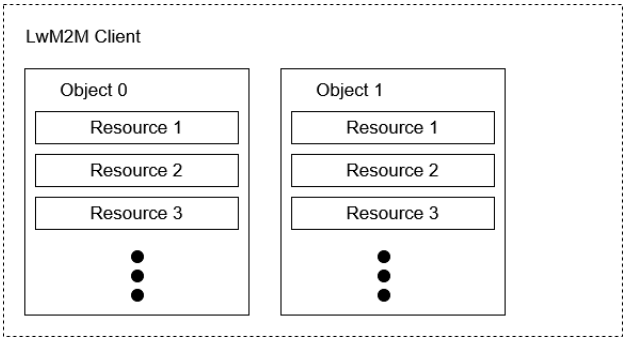

Figure 5: LwM2M objects, adapted from [16].

LwM2M defines application-level abstractions, which are called LwM2M Objects. Figure 5 shows the LwM2M Object Model. An LwM2M Object is a collection of individual resource definitions, each of which refers to a piece of information or system component like temperature value and battery level. Figure 6 shows an example LwM2M Location Object. The LwM2M Servers are allowed to read, write and execute the resources. However, the permissions for these operations are defined by the LwM2M Access Control Object which is set during the bootstrap.

The role of the LwM2M Server is to manage and control LwM2M Clients, and the role of the LwM2M Bootstrap Server is to provision the LwM2M Clients with the required credentials and connectivity information for the LwM2M Server. Typically, the credential is an LwM2M Server account for LwM2M Client, which might be in form of a pre-shared key (PSK), raw public key (RPK) and X.509 certificate. This provisioning step is called bootstrap. LwM2M supports four different types of bootstrap: factory bootstrap, bootstrap from smartcard, client initiated bootstrap, and server initiated bootstrap.

Factory Bootstrap: During manufacturing in the factory, the LwM2M Clients are provisioned with the required configuration such as a certificate, PSK and LwM2M Server connectivity information.

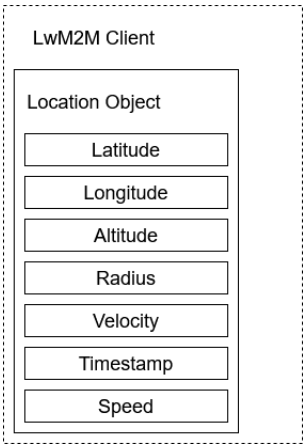

Figure 6: An example LwM2M Location Object, adapted from [16].

Bootstrap from Smartcard: If the device supports smartcard, then the bootstrap information is received from the smartcard. In case of smartcard removal and disablement, the bootstrap information is deleted from the client.

Client Initiated Bootstrap: The LwM2M Client initiates a connection to the LwM2M Bootstrap Server and retrieves the required credentials and connectivity information for the LwM2M Server. This mode requires the LwM2M Client to have a preloaded LwM2M Bootstrap Server Account in the device. The system that is designed in this thesis is based on the Client Initiated Bootstrap.

Server Initiated Bootstrap: A registered LwM2M Client can trigger server initiated bootstrap operations. For server initiated bootstrap, there already must be a connection between the LwM2M Client and Server to enter this mode.

2.1.3 Datagram Transport Layer Security

Datagram Transport Layer Security (DTLS) is a secure data transfer protocol which is used to protect data over datagram protocols (typically UDP). It provides an authenticated, confidentiality and integrity protected communication channel [17]. DTLS is an adaptation of TLS to provide same level of security for stateless connection protocols such as UDP [18]. DTLS 1.0 was introduced in RFC 4347 [19] by IETF in 2006. Then, it was superseded by DTLS 1.2 in RFC 6347 [18] in 2012. As of July 2020, DTLS 1.3 is still in draft [20]. The proof-of-concept (PoC) prototype that is implemented in this thesis uses DTLS 1.2.

DTLS is designed to work over unreliable transport channels, unlike TLS. TCP is slow due to its connection-oriented architecture. When data transfer speed is more important than reliability, or the protocol needs to be lightweight, DTLS is typically preferred over TLS. TLS cannot tolerate data loss and arrival of out of sequence data packages. Therefore, DTLS is preferred over TLS for IoT applications since it is less resource-intensive and does not need a large buffer to hold packages for reordering.

DTLS mainly adapts the TLS to solve the aforementioned limitations for stateless connection protocols while providing the level of security. Figure 7 shows the details of the DTLS handshake. It is important to comprehend the DTLS handshake to understand the proposed system in this work. This work extends the part of the handshake which authenticates the client. Section 3.5 explains this extension in detail.

Due to the connectionless architecture of UDP, DTLS is vulnerable to DDoS attacks [21]. Therefore, DTLS tries to mitigate the possible threats using cookie. The cookie should be generated from the connection parameters, such as client IP (Internet Protocol) address and port, and with a pre-defined random value. The DTLS handshake and cookie usage are as follows:

1. The client sends *ClientHello* message to the DTLS server to initiate a DTLS connection.
2. The server responds with a *HelloVerifyRequest* message, which contains a cookie in such a way that it can be verified without any per-client state on the server [18]. The cookie may be the hash of the concatenated string of the client IP address, port, and a pre-defined random value. The random value must be secret, and the client must not know it.
3. After the client gets the *HelloVerifyRequest* message, it resend the *ClientHello* message with the cookie. The DTLS server should not allocate any resource before getting a *ClientHello* message with a valid cookie.
4. After getting the valid *ClientHello* message, the server sends *ServerHello*, *ServerKeyExchange*, *CertificateRequest* and *ServerHelloDone* messages.
5. The client sends *Certificate*, *ClientExchange*, *CertificateVerify*, *ChangeCipherSpec* and *Finished* messages. The client sends the *Certificate* message if the server asks for the client certificate with *CertificateRequest* message.
6. The server sends *ChangeCipherSpec* and *Finished* messages, and the connection is established between the client and server.

This cookie mechanism mitigates the denial of service attacks and amplification attacks because the *ClientHello* message is longer then the *HelloVerifyRequest* only as the size of the cookie.

The messages marked with * in Figure 7 are optional, and they depend on the server's credentials. The *CertificateRequest* message is sent to client in order to ask for the client certificate. When the client gets this message, it replies with the client certificate. Later, the client certificate is used to authenticate the client. DTLS supports mutual authentication. However, usually only the server is authenticated. In the system that is designed in this thesis, we only authenticate the client based on its certificate and the out-of-band (OOB) channel message. The messages in the handshake have to be received and delivered in order. DTLS can handle messages which arrive out of sequence after the handshake is completed [18].

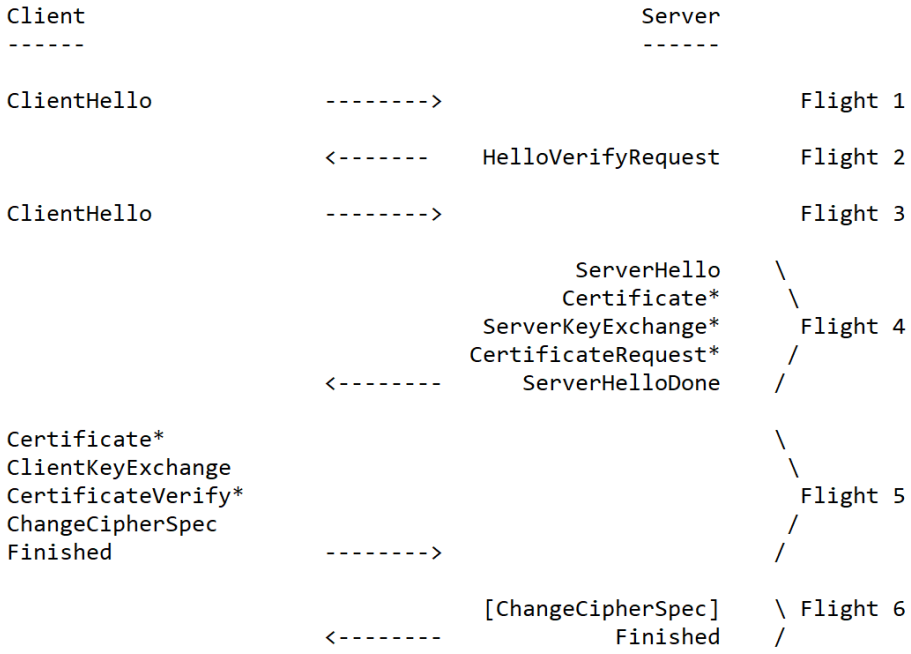

Figure 7: DTLS handshake steps in DTLS 1.2 [22].

2.2 Industry Solutions

2.2.1 Apple HomeKit

Apple HomeKit is a platform which enables smart home devices to be connected and control from iOS and macOS devices. It provides appliance manufacturers a platform to design and produce HomeKit-enabled devices. In a home with HomeKit, the smart devices can communicate with each other easily. The devices can be collected under a group so that devices can be controlled together in each room or in the entire house [23].

Apple HomeKit primarily uses a proprietary MFi (made for iPhone/iPod/iPad) authentication co-processor to authenticate the devices. The authentication co-processor allows the device to work within HomeKit framework. The authentication co-processor contains a cryptographic proof (a key-pair and a certificate) which can be used as a mechanism to prove that the device is a licensed device approved by Apple to be used with Apple devices. An accessory manufacturer that wishes their devices to work with HomeKit must obtain the MFi certification process. Also, Apple started to support authentication without the co-processor after iOS 11.3 release.

The HomeKit devices are paired with an iOS device which acts as the admin controller. The pairing procedure relies on OOB channel data transfer of an 8-digit pass-code. Along with the OOB channel, also the authentication co-processor is used depending on the device certification. The OOB channel exists in 2 forms: scanning a QR code on the device and transferring the 8-digit code manually from the device to the admin controller. After the OOB channel data transfer, the devices executes the

Secure Remote Password (SRP) protocol. Both of the devices derive long term keys and they exchanges the keys with key exchange protocols. Then, the devices uses the long term keys to authenticate each other and generate session keys.

Apple HomeKit is a closed and tightly-controller ecosystem. Also, it is not an open standard like LwM2M. Thus, Apple HomeKit is not reviewed by researchers as rigorously as open standards.

2.2.2 Xiaomi IoT Cloud

Xiaomi IoT Cloud is an ecosystem which consists of an IoT cloud system, a gateway device, and connected IoT devices attached to the gateway. The gateway provides a connection between the devices which do not support WiFi and the IoT cloud system. Usually, the devices which support Zigbee are connected to the gateway, and the gateway is connected over WiFi and the internet to Xiaomi Cloud. Other devices which support WiFi are directly connected to Xiaomi Cloud over WiFi and the internet. The connected devices are controlled with a smartphone [24]. Figure 8 shows the Xiaomi IoT Ecosystem.

Each Xiaomi IoT device is equipped with a unique Id and 128-bit Advanced Encryption Standard (AES) key [24]. The key is static, and it is not updated or provisioned during pairing. Initially, IoT devices which support WiFi are paired with the phone as shown with dashed lines in Figure 8. During the pairing process, each device is provisioned with a unique token, and it is used to authenticate the device to the cloud. After the pairing, the devices establish a connection to the cloud or to the gateway device depending on their wireless protocol support. Then, the controller device sends the command and control messages over the internet to the cloud services. Hence, the messages are passed through the cloud. The controller provisions required connectivity information and credentials. Then, the controller devices sends only control messages to the IoT devices through the cloud.

In 2018, several security vulnerabilities were found on some Xiaomi IoT devices by Dennis Giese, such as data residual after factory reset and poorly generated 256-bit encryption keys which have much less than 256-bit entropy [24]. If the credentials are not generated properly, it may cause the system to be vulnerable to brute force attacks. The credentials must be generated with high entropy as described in Section 3.2.1 to mitigate such vulnerabilities.

2.2.3 Samsung SmartThings

The SmartThings platform was initially introduced as an open-source platform in 2012 [25]. Later, the company which owns the SmartThings platform was acquired by Samsung, and the platform has evolved to a vendor-based model. The platform connects IoT devices through a hub device (also called home controller or gateway), which is controlled by client and cloud applications [26].

The smart devices are paired with the hub device, which supports communication protocols such as Zigbee, Z-Wave and Bluetooth. The connection between Z-Wave

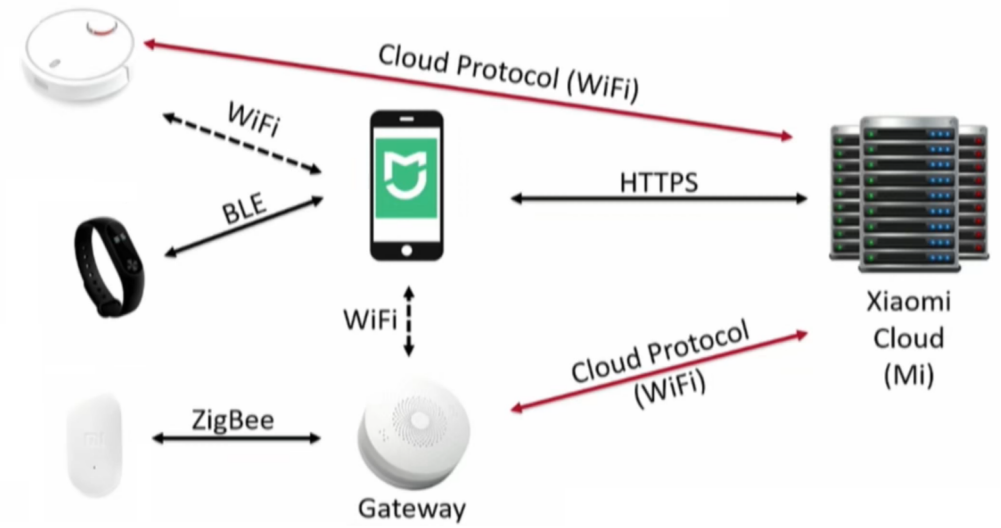

Figure 8: Xiaomi IoT Ecosystem, adapted from [24].

smart devices and the hub are secured by 128-bit AES encryption [27]. During the pairing, Z-Wave uses the Elliptic Curve Diffie-Hellman (ECDH) key exchange algorithm. The public keys of the smart devices are verified over an OOB channel. The OOB channel can be either scanning a QR code on the device or typing the first 5 digits of the decoded text in the QR code. The first digits of the QR code message are obfuscated during the wireless transmission. This mitigates the possibility for man-in-the-middle impersonation of devices during the pairing process. An example QR code is shown in Figure 9. The encoded text is: 34028-23669-20938-46346-33746-07431-56821-14553. The text includes 40 digits, which represent 128-bits of data. The underlined first 5 digits are used for manual typing. In this pairing system, the manual typing is less secure compared to the QR code scanning, because the verification of the device public key is based on a 5 digit number instead of a 40 digit number.

34028-23669-20938-46346-33746-07431-56821-14553

Figure 9: Example QR code for SmartThings.

2.3 QR Code

Quick Response code (QR code) is a 2-dimensional data matrix that encodes text and binary data. QR code was invented in Japan in 1994 to track automobile parts [28]. They have become popular also outside the automotive industry due to fast readability, low production cost, and greater storage capacity compared to standard barcodes. Nowadays, they are widely adopted and deployed for various purposes, such as advertisements, mobile payment and access control [28, 29].

A QR code consists of black squares arranged in a square grid on a white background which can be read by an imaging device like a camera. The encoded data is then extracted from patterns that are present in both horizontal and vertical components of the image [28]. A QR code can encode data up to 4296 alphanumeric characters [28].

QR codes support several error correction ratios. The supported error correction ratios are 7%, 15%, 25% and 30% [28]. Roughly speaking, a QR code which is created with 7% error correction ratio can be read even if 7% of its area is damaged. The bigger error correction ration means the bigger QR code size.

QR codes are vulnerable to modification and replacement attacks. Slight modifications on the QR codes can go unnoticed by humans since QR codes are not a human-readable format. Furthermore, an attacker can replace a legitimate QR code with a malicious one or paste the malicious one over the legitimate one [29]. This attack is easy to perform if the QR code is deployed in a form that is easy to remove such as a sticker. Krombholz et al. [29] proposed digital signatures to mitigate these attacks. While reading the QR code, the digital signature is also read. Then, this signature can be verified by trusted root certificates. However, this will increase the size of the QR codes due to the overhead of the signature, and it would require each QR code to be signed by a certificate authority. Note that the signature in the QR code mitigates only modification attacks. An attacker can still replace a signed QR code with another QR code with a valid signature.

3 System Description

3.1 Lifecycle of IoT Devices

This section describes the target system. This section is based on Huawei internal document [30].

The lifecycle of IoT device can be thought of having two modes: factory mode and operational mode, as Figure 10 shows. In the lifecycle, there are procedures that change the mode of the simple device: manufacturing, bootstrap, and factory reset procedures. The procedures are explained in Section 3.2

Factory mode: In the factory, the manufacturer builds the device. Then, the device manufacturer injects initial bootstrap credentials and trust anchors (see Section 3.4) for verifying peers into the simple device. The simple device starts its lifecycle in the factory mode when it leaves the factory. In this mode, the device is ready to be taken into use via the bootstrap procedures.

Operational mode: A simple device in the operational mode connects and communicates with a controller securely using operational credentials (see Section 3.4). Once the simple device is in the operational mode, it can be controlled by the controller device. The simple device continues to remain in this mode unless the factory reset procedure is triggered.

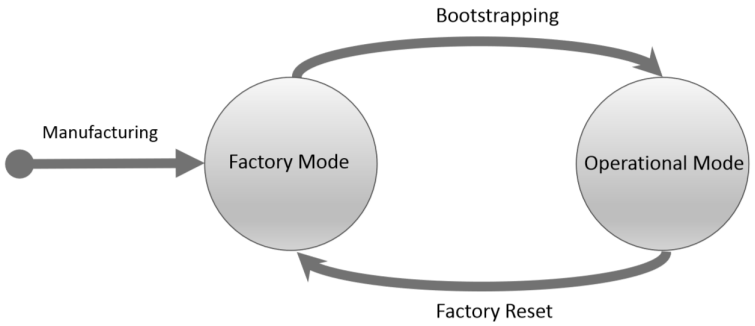

Figure 10: Lifecycle modes of a simple device.

3.2 Lifecycle Procedures

3.2.1 Manufacturing Procedures

During the manufacturing process, bootstrap credentials may be injected in the simple device. The bootstrap credentials may be a shared key with an identifier, or a device certificate and a corresponding key pair. The bootstrap credentials may also include metadata such as how to discover the device for bootstrapping.

The process to generate these credentials and the process to inject them on simple devices must be secured to prevent credential leakage. For example, a 128-bit encryption key must have 128-bits of entropy. The key should not be generated

by hashing a low-entropy alphanumeric text. The injection phase may also include additional steps where a reference to the bootstrap credentials is made available. For instance, this may be a QR code containing a hash of the device certificate, or the shared key may be printed in an additional leaflet to be included in the packaging.

The process of injecting bootstrap credentials are specific to manufacturers. Therefore, it is not further discussed in this thesis.

3.2.2 Bootstrap Procedures

The bootstrap procedure involves interaction between the simple device and the controller device. In this procedure, the controller device provisions operational credentials to the simple device. The operational credentials are used by the simple device to authenticate itself to other simple devices and the controller.

Trust roots are used by the simple device to authenticate other simple devices and the controller device. The operational credentials contain either an asymmetric key pair and a certificate, creating a binding between the public key and the simple device identity, or a symmetric key and key identifier. Either of the key types can be generated in the simple device, or they may be generated by the controller and transferred to the simple device during the bootstrap procedures. If the key is an asymmetric key pair, a certificate is issued to the public key of the key pair by the controller. The certificates issued to devices and trust roots together form an end-user specific public key infrastructure (PKI), in which the end user's devices can authenticate each other. If the key is a symmetric key, a unique key identifier is created by the controller.

The bootstrap procedures consist of three phases:

Out-of-band phase: Simple-device specific data is transferred to the controller via an out-of-band channel. The transferred data is used in the discovery and bootstrap phase. The structure of the OOB channel message is explained in Section 4.3.

Discovery phase: The controller discovers the simple device and then connects to it.

Bootstrap phase: The controller performs the bootstrap procedures over a secure channel, including the provisioning of the operational credentials, which has been established after the discovery phase.

3.2.3 Factory Reset Procedures

The factory reset procedure returns the simple device from the operational mode back to the factory mode. This procedure wipes all the operational credentials and operational data stored on the simple device. The bootstrap credentials that were injected during the manufacturing process are retained since these credentials are required for bootstrap procedures when the device is reused. The factory reset can be triggered by a button on the device or remotely via controller.

3.3 Architecture

Figure 11 shows an overview of the system. The designed system consists of two main parts: simple device and controller. This thesis focuses on the simple device part of the system. The details of the controller part are the subject of another master thesis that was done parallelly in the same project [11]. The controller details is the topic of another master's thesis [11] in the same project. The controller and simple device presents three interfaces: bootstrap, management, and usage.

The controller device have three entities that are responsible for carrying out the procedures: bootstrap, management, and usage. Note that it is not necessary that all 3 entities reside in the same physical device. The controller entities could be distributed to different locations. For example, the Bootstrap Entity could be in a smartphone, while the management and usage entities are in a remote server. However, the communication and coordination between the distributed entities of the controller are not in the scope of this thesis.

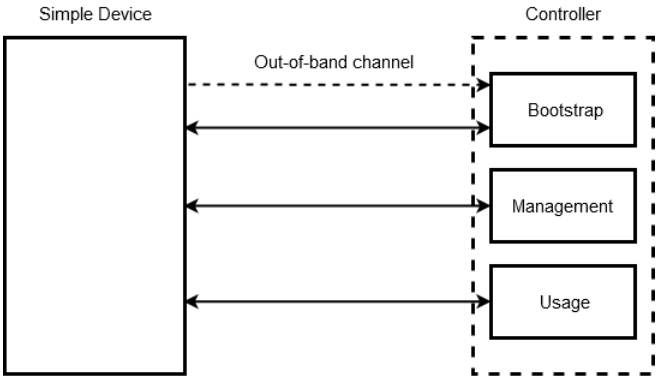

Figure 11: Overview of the system [30].

3.3.1 Entities

Bootstrap Entity: This entity is responsible for the bootstrap procedures of the simple device in the controller when the device is in the factory mode. It uses the bootstrap interface and an OOB channel communication.

Management Entity: This entity uses the management interface to control the simple device when the device is in the operational mode.

Usage Entity: This entity accesses the services offered by the simple device when the device is in the operational mode via the usage interface.

3.3.2 Interfaces

Bootstrap interface: It is used to provision the simple device with operational credentials and trust roots (e.g., list of root certificates, or shared keys). The operational

credentials are used by third parties (controller devices and other devices) to authenticate the simple device, and trust roots are used by the simple device to authenticate the third parties. The third-party entities are the controller entities. The bootstrap procedure can be initiated by the simple device if the simple device is in the factory mode. Upon successful completion of the bootstrap procedure, the simple device switches to the operational mode.

Out-of-band interface: It is used to provide an additional authentication mechanism to increase the security of the bootstrap interface. It can also be used to enhance the discovery mechanism between the simple device and the Bootstrap Entity before the execution of the bootstrap procedure.

Management interface: It is used to manage the simple device. The management procedures may include the possibility to add, change, and remove operational credentials and trust roots. The management procedures may also include the possibility to initiate the factory reset procedure. Management procedures can be performed with the simple device if the simple device is in the operational mode.

Usage interface: It is used to use and operate the simple device. The user operations include any operations that are associated with the features of the simple device. User operations can be performed with a simple device if the simple device is in the operational mode.

3.4 Types of Credentials

3.4.1 Bootstrap Credentials

The simple device is provisioned with bootstrap credentials during the manufacturing of the device. The bootstrap credentials are simple-device specific and should be unique. The bootstrap credentials can be used during the bootstrap procedure to increase the security of the bootstrap procedure. The bootstrap credentials can be the following:

Symmetric key: A symmetric key is typically conveyed to the controller device via out-of-band means to provide the authentication of the simple device.

Asymmetric key pair: The hash of an asymmetric key pair's public key is typically conveyed to the controller device via out-of-band means to provide the authentication of the simple device. The asymmetric key might be a raw public key (RPK).

Asymmetric key pair and device certificate: The information may be conveyed to the controller device via out-of-band means to provide the authentication of the simple device. The device certificate provides additional means to authenticate the device remotely, and it can also be used to provide other services like attestations. The device certificate might be X.509 certificate.

3.4.2 Operational Credentials

The operational credentials are provisioned to the simple device during the bootstrap procedure. The operational credentials may include but are not limited to:

Asymmetric key pair: The asymmetric key pair is used by the simple device to authenticate itself to another device. The asymmetric key pair is generated during the bootstrap procedure either by the simple device or the controller. In the former case, the simple device requests a certificate for the key pair generated by the device from the controller device, and in the latter case, the key pair is transferred to the simple device during the bootstrap procedure together with the certificate for that key pair.

Certificate: The certificate is used to bind an identity assigned by the controller to the asymmetric key pair of the operational credentials. The certificate is issued by the controller device which is capable of issuing certificates for simple devices. The certificate can be generated by a user-specific certification authority, or it can be generated as a self-signed certificate by the controller. The certificate is used together with the asymmetric key pair in any PKI-capable protocol such as DTLS and TLS.

Symmetric key: The symmetric key pair is used by the simple device and another device either directly or indirectly to authenticate the simple device to the other device. The symmetric key is generated during the bootstrap procedure either by the simple device or the controller. Both the simple device and the controller store securely the symmetric key and associate it with the key identifier.

Key identifier: The key identifier is used to identify a symmetric key. The key identifier is created during the bootstrap procedure by the controller. The controller must assign a unique key identifier to each generated symmetric key value.

Trust roots: The trust roots are used as roots of trust for any certificate validation process, i.e., they are used to authenticate other devices and entities in any PKI-capable protocol. Trust roots can also be used to authenticate other devices and entities using a shared symmetric key. In this case, the trust roots contain a list of <symmetric key, key identifier> pairs of other devices.

3.5 Adaptation of LwM2M

The LwM2M specification requires that the LwM2M Client is authenticated to the LwM2M Server and LwM2M Bootstrap Server. This can be accomplished with a pre-shared key or public key infrastructure. However, in the cases where a PKI does not exist, authentication is a major problem. Furthermore, PKI helps the controller to authenticate the simple device; however, it does not help the controller to uniquely identify a simple device. Also, the controller device needs metadata to discover the simple device during the bootstrap. Therefore, this thesis proposes the use of OOB channel communication to transfer the metadata to enable the discovery of the simple device by the controller and information about the simple device credentials to uniquely identify each simple device. To transfer this information, this thesis

proposes the use of QR code scanning as the OOB channel communication. LwM2M bootstrap sequence was modified to include the OOB communication.

Figure 12 shows the bootstrap steps for a standard LwM2M Client and LwM2M Bootstrap Server. The steps are as follows:

1. The LwM2M Client initiates a DTLS connection to the LwM2M Bootstrap Server.
2. The LwM2M Bootstrap Server authenticates the LwM2M Client during the DTLS channel establishment based on the credentials of the LwM2M Client. The credential might be a PSK, an RPK, or an X.509 certificate.
3. The LwM2M Client initiates the bootstrap sequence.
4. The LwM2M Bootstrap Server sends the LwM2M Server Object and LwM2M Security Object to the LwM2M Client. Other objects such as WLAN Connectivity information can optionally be sent if the LwM2M Client needs them. The LwM2M Bootstrap Server can determine whether the client needs the extra information by checking the device type and in-band channel type. The device and in-band channel type are indicated in the OOB channel message (see Section 4.3).
5. The LwM2M Bootstrap Server sends the Bootstrap-Finish message.
6. After completing the bootstrap, the client closes the LwM2M Bootstrap Server connection. Then, the client initiates a new DTLS connection to the LwM2M Server with the credentials and the connection information in the LwM2M Server Object which received during the bootstrap. Then, the LwM2M Client registers to the LwM2M Server.

In the proposed adaption in this thesis, simple device takes the role of LwM2M Client, and the controller (Bootstrap Entity) takes the role of LwM2M Bootstrap Server. In the adaption, the bootstrap steps are as follows:

1. The LwM2M Client opens a wireless access point.
2. The controller scans the QR code on the simple device and then connects to the wireless access point which is hosted by the simple device.
3. The LwM2M Client initiates a DTLS connection to the LwM2M Bootstrap Server when the controller device connects to the access point. The access point will probably have a Dynamic Host Configuration Protocol (DHCP) server; thus, the connection can be triggered by an IP lease trigger. The DHCP server leases only one IP, and it is the bootstrap server IP address.
4. The LwM2M Bootstrap Server authenticates the LwM2M client during the DTLS channel establishment. For example, if the LwM2M Client uses X.509 certificate or RPK, the controller can authenticate the client by comparing the hash of the client credential and the hash in the OOB message. If the client uses PSK, the controller can authenticate the client with the PSK in the OOB

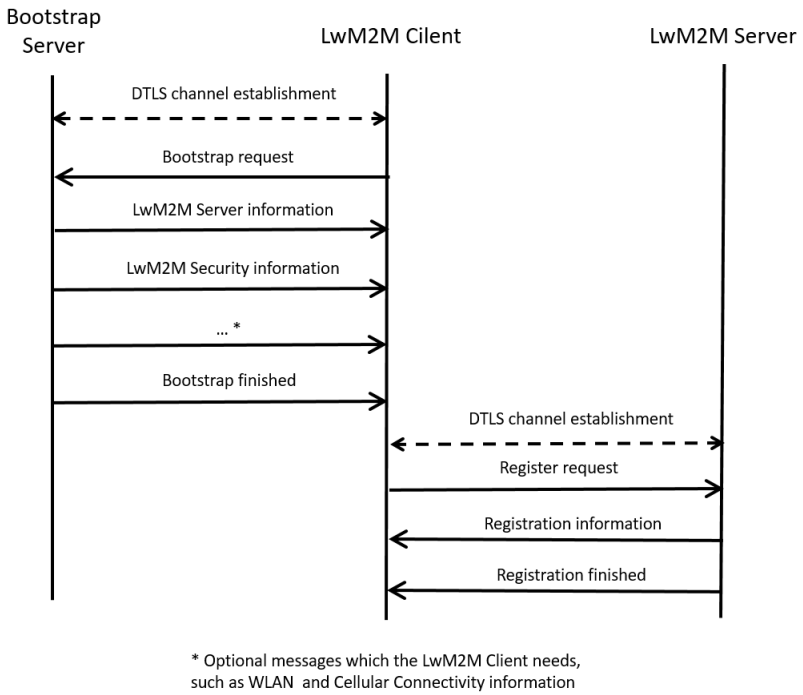

Figure 12: Bootstrap steps.

message. Figure 13 shows the authentication steps for X.509 certificate based authentication during the DTLS handshake and the bootstrap procedures.

5. The LwM2M Client initiates the bootstrap sequence.
6. The LwM2M Bootstrap Server sends the LwM2M Server Object and LwM2M Security Object to the LwM2M Client. Other objects such as WLAN Connectivity can optionally be sent if the LwM2M Client needs them. The LwM2M Bootstrap Server can determine whether the client needs the extra information by checking the device type. The device type can be deduced from the device's account identity (ID) with a type indicating prefix which is concatenated to the account ID.
7. The LwM2M Bootstrap Server sends the Bootstrap-Finished message.
8. After the bootstrap is completed, the device closes the LwM2M Bootstrap Server connection and the access point. Then, it connects to the home WiFi network.
9. The client initiates a new DTLS connection to the LwM2M Server with the credentials and the connection information in the LwM2M Server Object which received during the bootstrap. Then, the LwM2M Client registers to the LwM2M Server.

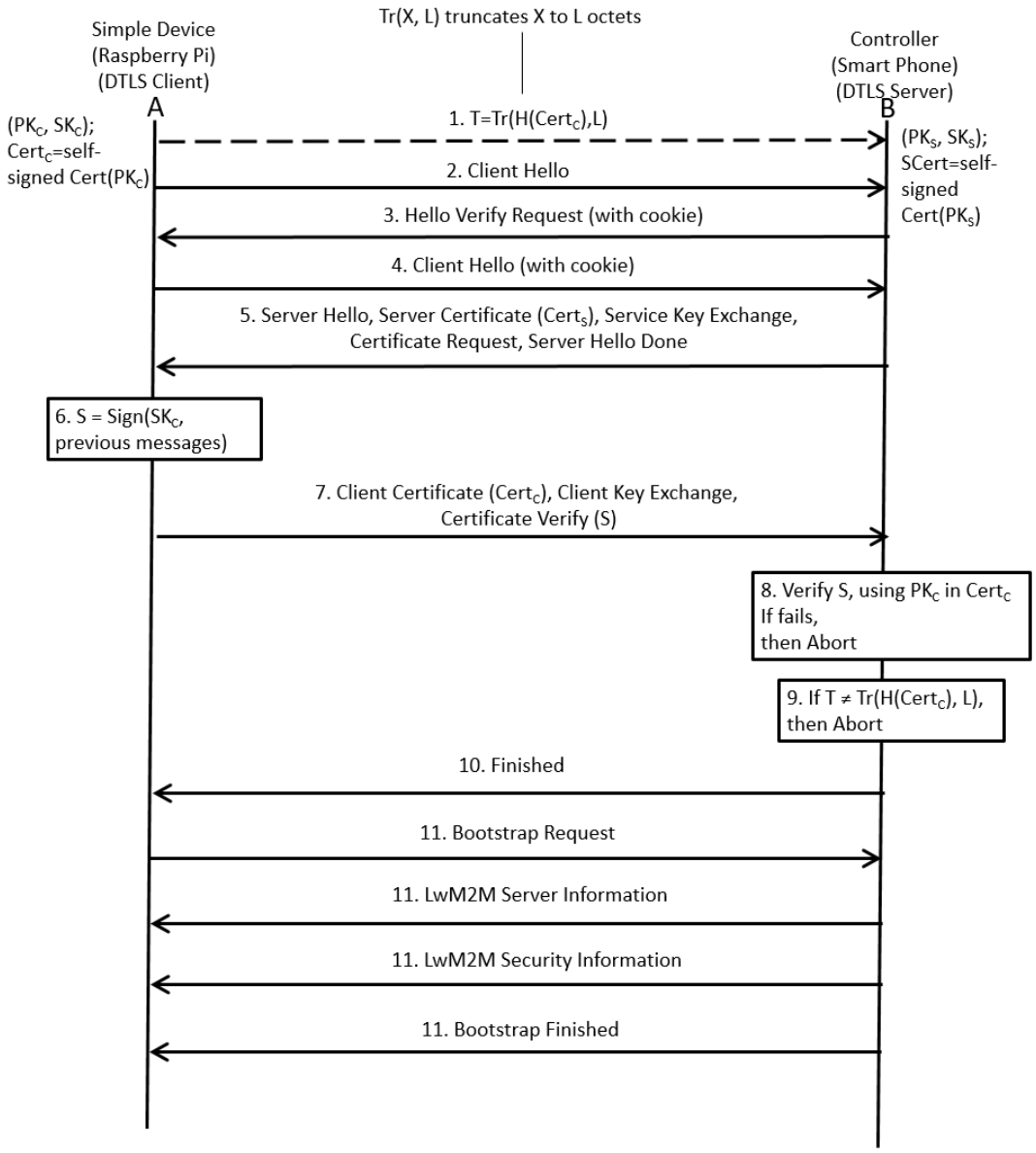

Figure 13: Authentication steps.

3.6 Security

The lifecycle procedures change the lifecycle mode of the simple device. Therefore, the procedures have a crucial effect on the lifecycle of IoT devices, and they should be carried out by authorized parties.

The OOB channel message security has a remarkable impact on the security of

the bootstrap procedures. Most OOB channels including QR code scanning are vulnerable to eavesdropping [31]. Several OOB channels are susceptible to message tampering or spoofing. This opens a vulnerability to impersonation and man-in-the-middle (MitM) attacks. There should be a mechanism to prevent or detect modifications of the printed QR code e.g., the QR code might be printed on the device with permanent ink.

The length of the certificate hash in the OOB channel message affects the system security. The length should be defined according to the relevant standards such as National Institute of Standards and Technology (NIST) Recommendation for Applications Using Approved Hash Algorithms.

During the factory reset procedures, the operational credentials and any user data must be securely wiped from the device. Data residue in the device may cause privacy issues like leaking sensitive information about the user. The simple device can use secure key storage to keep the keys safe. Also, the memory area which holds the user data can be encrypted with a key that is stored in a secure hardware module. Encrypting the user data and storing the key in a secure hardware provides protection for the stored user data. Simply wiping the encryption key securely is enough to make the user data unrecoverable.

The LwM2M specification supports different credential types; no security, PSK, RPK, and X.509 certificates. No security and RPK should be supported as they are mandated by the LwM2M standard [10]. However, it is recommended that the no security option should be used with other security protocols like OSCORE. Not using encryption (i.e., DTLS, TLS) means that the data is transferred as plain text. An unencrypted and non-integrity-protected connection is vulnerable to eavesdropping and alteration. Therefore, when encryption is not used, the other security protocols should be used to secure the communication.

4 Implementation Details

This section explains the details of the proof-of-concept prototype and explains the libraries and platforms that are used.

The PoC prototype is implemented with two Raspberry Pies and one Android smartphone. The system can be implemented with several different configurations. As stated in Section 3, the entities in the controller device can be distributed to different locations and devices. Figure 14, 17, and 20 show three different implementation configurations.

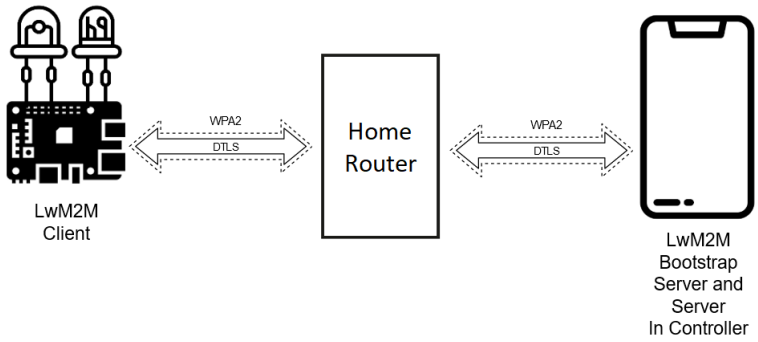

Figure 14: Configuration 1 overview.

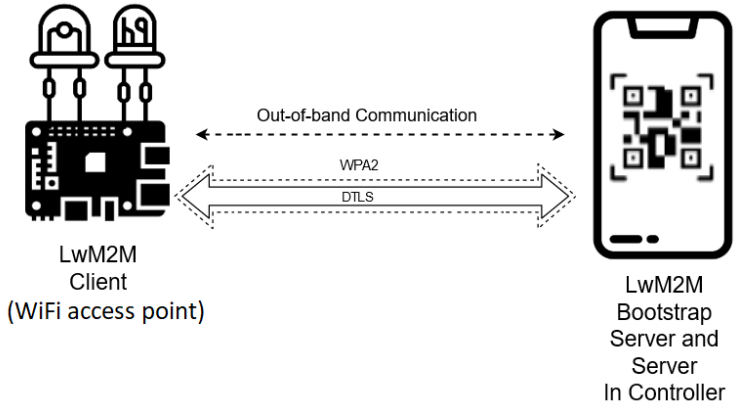

Figure 15: Configuration 1 during bootstrap mode.

Figure 14 shows the first implementation configuration. In the first configuration, the LwM2M Server and LwM2M Bootstrap Server are in the controller device. In the bootstrap mode, the LwM2M Client opens a WiFi access point and waits for the controller to connect to it. The controller scans the QR code on the LwM2M Client, then discovers the LwM2M Client. During the discovery phase, WiFi connection is established between the LwM2M Client and the controller. After the WiFi connection is established, DTLS connection is established between the LwM2M Client and the controller. Then, the bootstrap procedures take place between the controller and the LwM2M Client. Figure 15 shows the communication channels between the

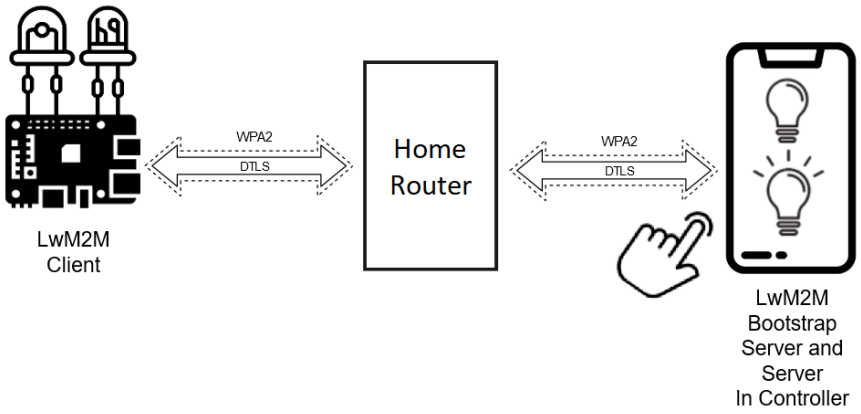

Figure 16: Configuration 1 during operational mode.

LwM2M Client and the controller during the bootstrap. After the bootstrap, the LwM2M Client closes the WiFi access point, and both the client and the controller switch to the home WiFi network. The LwM2M Client switches to operational mode after the bootstrap is completed. During the operational mode, the LwM2M Client and controller can send messages to each other over the home WiFi router. Figure 16 shows the communication channels between the client and the controller in the operational mode.

This configuration is easier to implement compared to the other configurations since the configuration does not require extra communication channels between the entities. However, due to the limitations on porting the Leshan LwM2M Server implementations to Android, this configuration was not implemented [11].

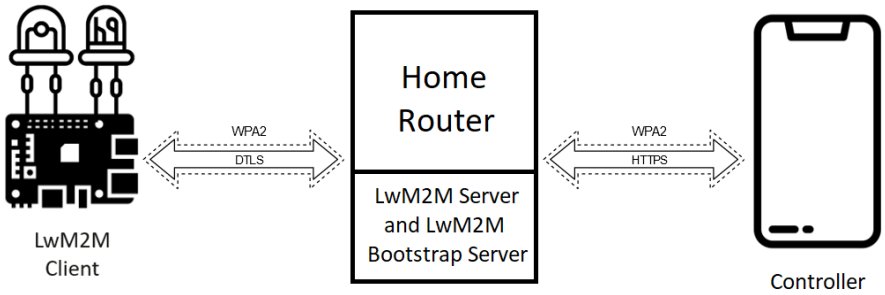

Figure 17: Configuration 2 overview.

Figure 17 shows the second implementation configuration. In the second configuration, the LwM2M Server and LwM2M Bootstrap Server are in a gateway device. The gateway device might be the home router or a separate device. In the second configuration, the LwM2M Client opens a WiFi access point. Then, the controller device scans the QR code on the simple device and transfers the decoded QR code message to the gateway device over HTTP Representational State Transfer (REST) Application Programming Interface (API). Figure 18 shows the communication channels between the gateway device, the controller, and the LwM2M Client. The

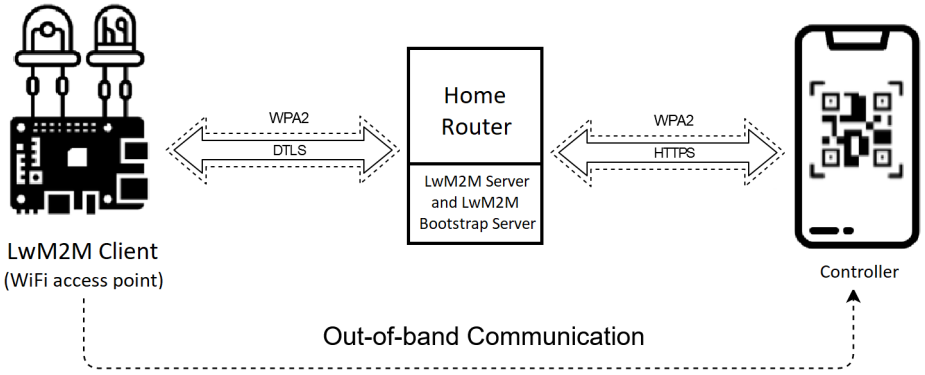

Figure 18: Configuration 2 during bootstrap mode.

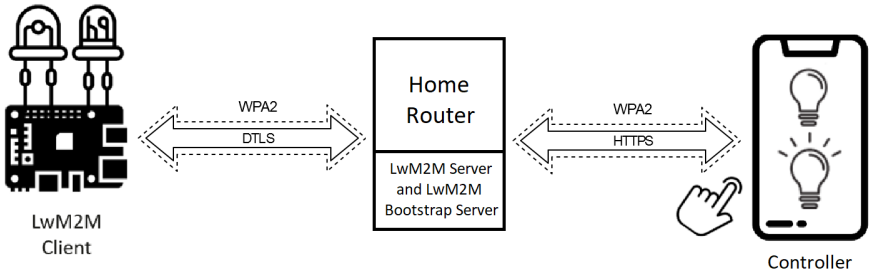

Figure 19: Configuration 2 during operational mode.

gateway device establishes WiFi and DTLS connection to the LwM2M Client. The bootstrap procedures take place between the gateway device and the simple device. After the bootstrap is completed, both the client and the gateway device switch to the home WiFi network. During the operational mode, the controller sends command and management messages over the HTTP REST API to the gateway device. Then, the gateway device converts the messages into LwM2M messages and sends them to the simple device over the DTLS channel. Figure 19 shows the communication channels between the gateway device, the client, and the controller during the operational mode. This configuration supports multiple controllers.

Figure 20 shows the third implementation configuration. In the third configuration, the LwM2M Server and LwM2M Bootstrap Server are in a cloud system. The cloud system may be provided by the device manufacturer. In this configuration in the bootstrap mode, the LwM2M Client opens a WiFi access point. Then, during the discovery phase, the controller scans the QR code, then it stores the decoded and sends the data to the LwM2M Bootstrap Server through the HTTP REST API in the cloud over the internet. The controller device either downloads the bootstrap information from the bootstrap server and sends the information to the client or acts as a message forwarder between the bootstrap server and the client. Figure 21 shows the communication channels in the bootstrap mode. The controller establishes WiFi and DTLS connection to the client. Then, it either sends the stored bootstrap information or forwards the messages from the bootstrap server in the cloud. After

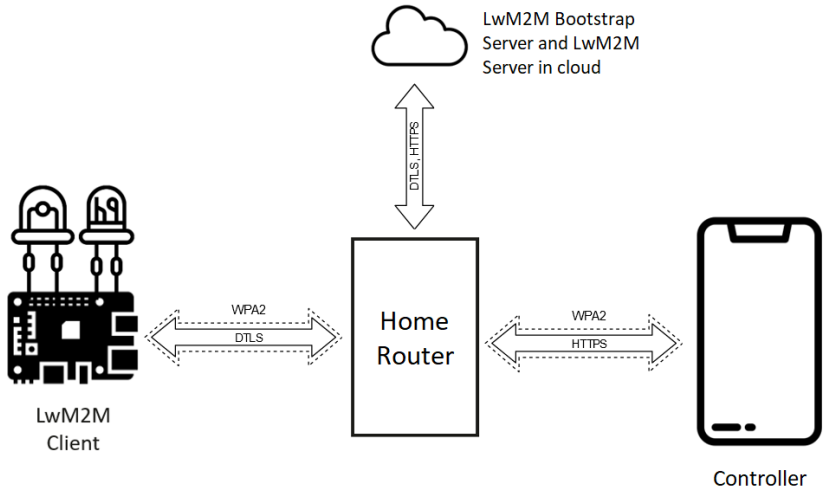

Figure 20: Configuration 3 overview.

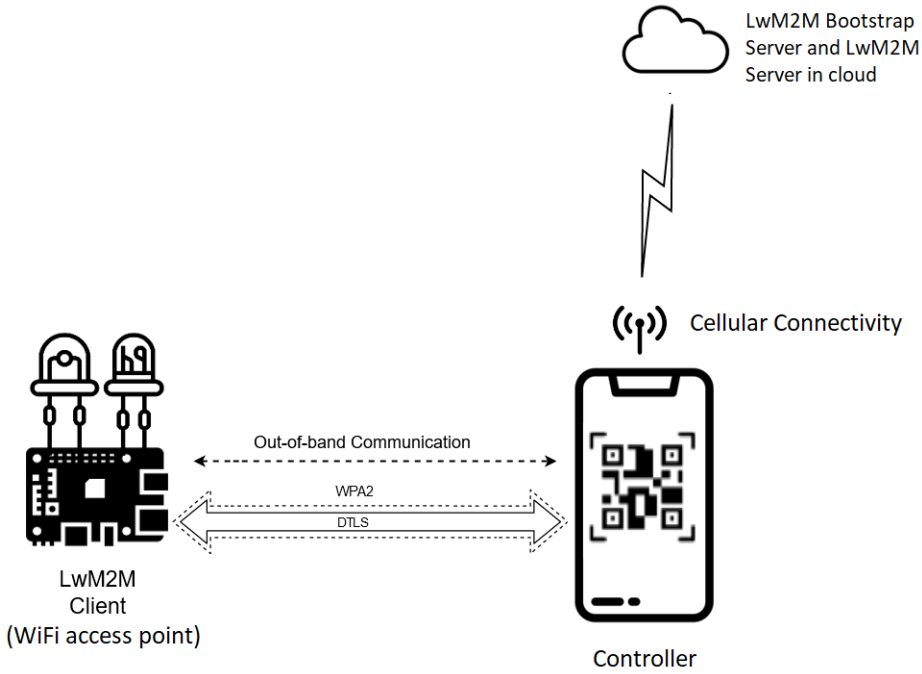

Figure 21: Configuration 3 during bootstrap mode.

the bootstrap is completed, the client closes the WiFi access point and switches to the home WiFi network. In the operational mode, the controller sends command and management messages to the LwM2M Server in the cloud through the HTTP REST API. The LwM2M Server converts the HTTP REST API messages into LwM2M messages and sends them to the client over the internet. Figure 22 shows the connections in the operational mode. In this configuration, the controller does not have to stay connected to the home WiFi network to control and manage the client since the LwM2M Server is accessible over the internet.

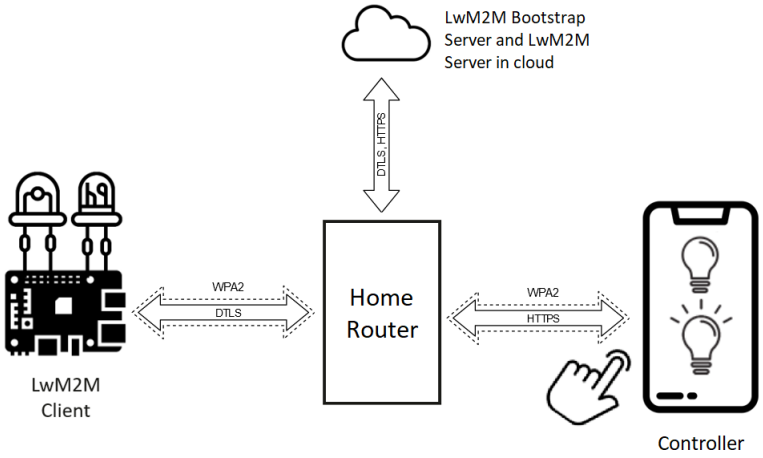

Figure 22: Configuration 3 during operational mode.

This configuration is possible because the bootstrap server is not authenticated in the bootstrap procedures. Also, the controller can act as a message forwarder if it supports Multipath TCP [32]. Multipath TCP enables a device to simultaneously connect to WiFi and cellular network. Thus, the controller device can receive messages from the cloud system and forward them to the simple device over the WiFi connection.

4.1 Platform Description

4.1.1 Raspberry Pi

Raspberry Pi is a low cost, single board computer family designed by Raspberry Pi foundation. The Raspberry Pi family has various models for different purposes. It supports different operating systems which are mostly based on Linux such as Raspbian, DietPi and RISC OS. This platform is widely used for PoC projects due to the low cost and community and library support. Therefore, Raspberry Pi 3 Model B+ was chosen for the PoC prototype. More IoT oriented platforms, such as Onion Omega2 and ESP8266, could be chosen; however, that platform lacks support and has compatibility problems with some software libraries.

The Raspberry Pi 3 Model B+ has a quad-core ARM Cortex-A53 processor inside a Broadcom BCM2837B0 system-on-chip (SoC) module. It has a Gigabit Ethernet module over Universal Serial Bus (USB) 2.0 and an on-board WiFi module, which supports Institute of Electrical and Electronics Engineers (IEEE) 802.11.b/g/n/ac modes. Raspberry Pi 3 Model B+ has 40 general-purpose input and output pins, which enable it to interact with peripheral devices like sensors and actuators [1]. These pins are used to control the light emitting diode (LED) and push button in our PoC prototype.

[1] https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus

4.1.2 DTLS Library

Since the Wakaama project uses the tinydtls library for the DTLS layer, it would be natural to also use tinydtls in our implementation. However, tinydtls does not support authentication based on X.509 certificates, and it supports only two cipher suites. During initial experiments, we found that Wakaama tinydtls integration supports only the shared-key authentication option, and that tinydtls is rather fragile: it crashed when another DTLS server implementation presented the client with unknown authentication options.

For those reasons, we looked into OpenSSL, GnuTLS, and MbedTLS as alternatives to tinydtls during the design. In the end, we chose MbedTLS, firstly because it is designed for resource-constrained embedded devices, and secondly, because there were people in the same project who were familiar with MbedTLS.

	X.509 cert.	**raw public key**	**pre-shared key**
tinydtls	✗	✓	✓
OpenSSL	✓	✗	✓
GnuTLS	✓	✓	✓
MbedTLS	✓	✗	✓

Table 3: Types of credentials in four cryptographic libraries.

Table 3 shows the types of credentials supported in those four cryptographic libraries. For full compatibility with CoAP, the DTLS library should support the TLS-PSK-WITH-AES-128-CCM-8, TLS-ECDHE-ECDSA-WITH-AES-128-CCM-8 cipher suites; raw public key; and the "no security" authentication options. Note that since MbedTLS lacks raw public key support, it does not fulfill all of requirements of CoAP. However, since authentication based on raw public key is a subset of the authentication based on X.509 certificates, it should not be difficult to add support for raw public key authentication to MbedTLS.

MbedTLS (previously know as PolarSSL) is an open-source TLS/DTLS/SSL (SSL stands for Secure Sockets Layer) library, which is developed by ARM. The library is coded in C. It supports Distinguished Encoding Rules (DER) and Privacy Enhanced Mail (PEM) encoded X.509 certificates and more than 150 cipher suites [2]. Supporting DER encoded certificates is important for the LwM2M protocol since all certificates are sent in DER format over the network. MbedTLS also supports static allocation of memory. Thus, it is suitable for resource-constrained devices, such as embedded devices and IoT devices.

4.1.3 WiringPi

WiringPi is a pin-based General Purpose Input/Output (GPIO) access library written in C for Raspberry Pi. It enables C programs to control GPIO pins, pulse width modulation (PWM) functionality on the pins, and to register interrupt service

[2] https://tls.mbed.org/supported-ssl-ciphersuites

routines. This library is used to control the LED brightness and to trigger factory reset on push button press. It requires root privileges to make changes to the pins. Thus, the program which uses this library has to be executed with root privileges.

4.2 Software Architecture and Components

In the PoC implementation, the second configuration was implemented (the other configuration details in Section 4). The PoC prototype is based on three main components: the simple device, the gateway device, and the controller. The simple device is Raspberry Pi 3 Model B+, which acts as an LwM2M client. The gateway device is Raspberry Pi 3 Model B+, which hosts the LwM2M Server and LwM2M Bootstrap Server. The controller is an Android smartphone, which hosts LwM2M Server and LwM2M Bootstrap Server. Figure 23 shows an overview of the PoC prototype. However, the bootstrap server is not necessarily required if the LwM2M Server information is provisioned during the manufacturing process.

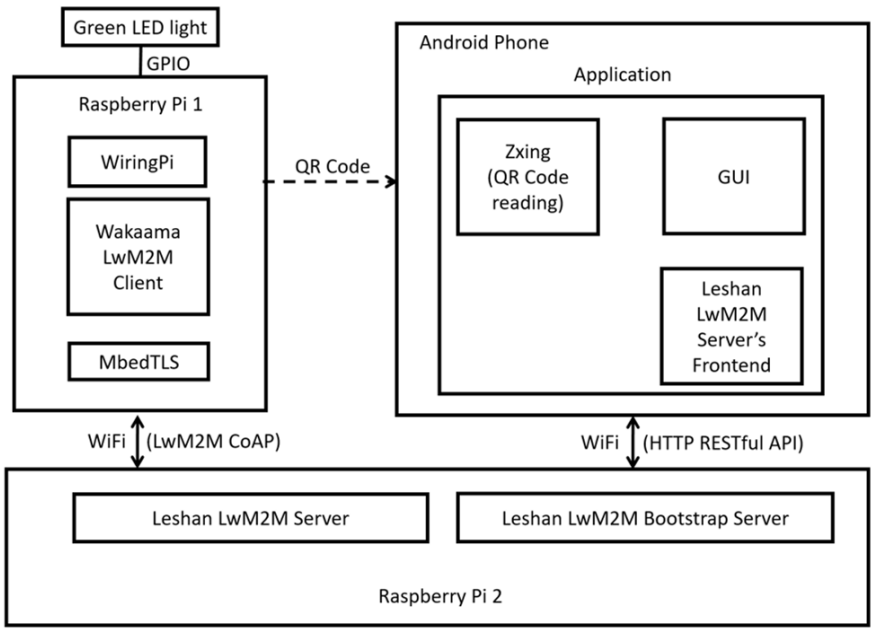

Figure 23: Overview of the PoC prototype

The core of the LwM2M Client is the Wakaama project. The Wakaama project is an open-source LwM2M Client implementation. The project uses several other open-source projects, such as Erbium CoAP and tinydtls [3]. Wakaama itself does not implement its own DTLS functionality; by default it utilizes tinydtls (a DTLS library for IoT devices [4]) to secure the connections. Tinydtls does not support X.509 certificates. It supports only PSK, RPK, and NoSec authentication mode. In the PoC implementation, X.509 certificate support was added to Wakaama by integrating it with MbedTLS (more details on MbedTLS in Section 4.1.2).

[3] https://github.com/eclipse/tinydtls
[4] https://projects.eclipse.org/projects/iot.tinydtls

The messages between the LwM2M Client and LwM2M Servers are transported with UDP over Internet Protocol (IP). The LwM2M Client and the LwM2M Servers are connected over WiFi network. LwM2M supports several other communication protocols, such as SMS and cellular IoT communication protocols. However, UDP is more suitable for home users. Figure 24 shows the software libraries which are used in the PoC prototype and the corresponding layers in the protocol stack.

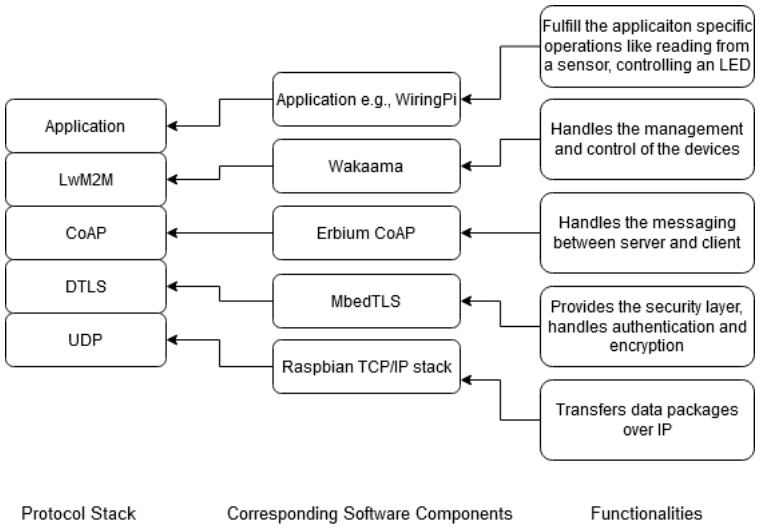

Figure 24: Software components

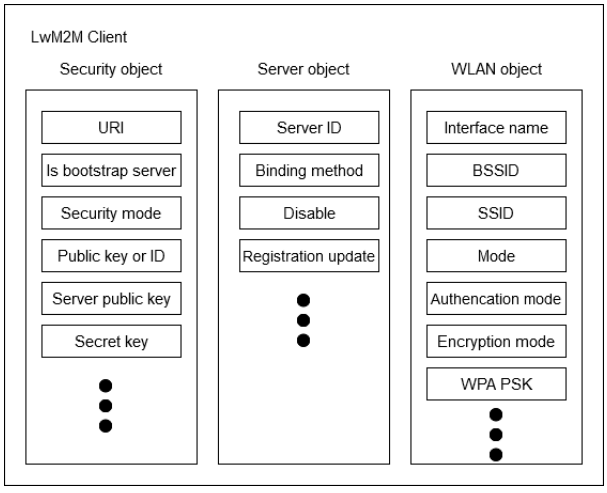

Figure 25: Persistent objects in the PoC prototype.

The LwM2M Objects consist of mandatory and optional fields. Depending on the application, the optional fields may need to be implemented. For example, in the PoC prototype, the LwM2M Client controls the brightness of the LED. The dim value of the LED is an optional field in the LwM2M Light Control Object.

The LwM2M Client implements seven LwM2M objects as Figure 26 shows. The security, server and device objects are mandatory to implement for an LwM2M Client;

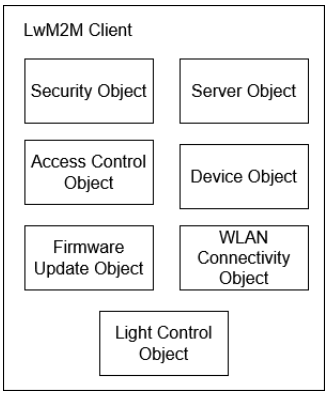

Figure 26: Supported objects in the implementation.

The other objects are optional. WLAN connectivity and light control objects are added to the Wakaama project for the PoC. The Wakaama project normally stores theses LwM2M Objects in RAM. After the device is rebooted or the program is restarted, all the objects are gone. To store the objects in the read-only memory (ROM), object storage functions were added. Figure 25 shows the objects that are stored in the file system. The security, server, and WLAN connectivity objects are stored as a binary file in ROM. The binary file format was chosen to save memory space and to comply with the LwM2M standard. LwM2M requires the clients to store the certificates in DER format, which is a binary encoding format for X.509 certificates.

Each layer and software library fulfills the functionality of the LwM2M Client. The client provides four interfaces: OOB, bootstrap, management, and usage. The OOB channel interface is based on providing a QR code that is printed on the retail box of the LwM2M Client device. The bootstrap, management, and usage interface are provided by the Wakaama project. Wakaama handles the state transitions, bootstrap procedures, and management operations. It dispatches the usage messages, such as turn on a light and turn off a light, coming from the controller to the relevant software libraries via function calls.

In the PoC prototype, the client opens a WiFi access point. The access point functionality is implemented with hostapd software.

4.3 Design of Out-of-Band Channel Message

In the PoC prototype, the OOB message is transferred via QR code scanning. QR code is scanned by the controller, which is an Android smartphone. The reasons for choosing this technique are as follows: First, QR codes are easy to create, compared to, e.g., near-field communication (NFC) tags or data-modulated audio signal. Second, smartphones typically already include an integrated camera.

This section describes the OOB channel message, shown in Figure 27, that is transferred from the simple device to the controller. This message is designed

to support several in-band communication technologies like WiFi and Bluetooth. However, only WiFi in-band channel is used in the PoC prototype. It also supports several types of device credentials including pre-shared key, raw public key, and X.509 certificate.

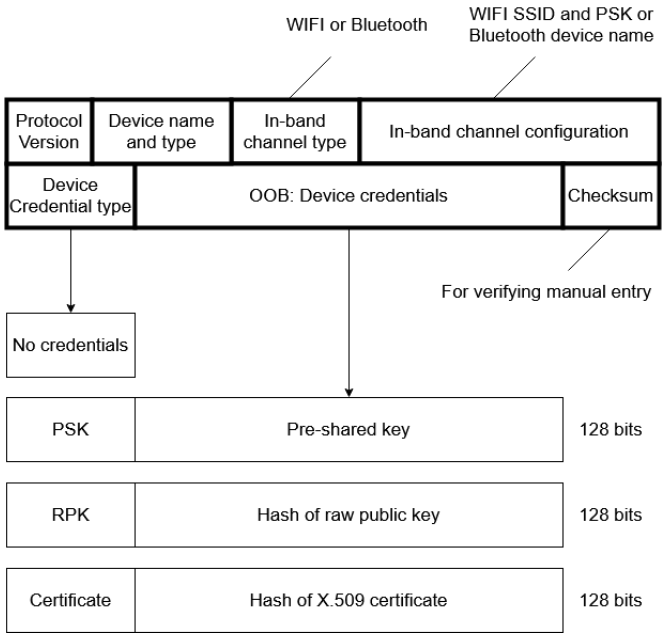

Figure 27: Structure of the OOB message.

The following example demonstrates the structure of the OOB channel message:

```
01;EP:055192;T:LED;WIFI;WPA2-PSK:huawei_iot:5bd1f120735d1ba4e2d2149ae
b891850;03;367f36ee1f5e272cfbca1ca35a73e42b;34;;
```

Each message field is separated with a semicolon (;), sub fields are separated with a colon (:), and double semicolons (;;) denote the end of the string. The message fields from left to right are as follows:

- 01: The protocol version for the OOB message structure; This field makes it easier to define new versions of the OOB message structure.
- EP:055192: The device name and type (EP stands for end point); It is used as identity for the LwM2M Server account. The device name is generated during the manufacturing, and it must be unique.
- T:LED: This field indicates the device type. The device name may be required for the LwM2M Servers to identify the device correctly and determine the extra information that the device needs during the bootstrap procedures.
- WIFI: The in-band channel communication protocol; In-band channel might be other communication protocols such as Bluetooth and Zigbee.
- 03: The device credentials type; Possible values of this field: 00, 01, 02, 03 are, respectively: no security, PSK, RPK and X.509 certificate.

- WPA2-PSK:huawei_iot:5bd1f120735d1ba4e2d2149aeb891850: The credentials for the in-band communication channel that is used during bootstrap, when the simple device acts as a WiFi access point. These three fields include the security algorithm (WPA2-PSK), Service Set Identifier (SSID) (huawei_iot), and the PSK (5bd1f120735d1ba4e2d2149aeb891850). These fields can be extended in the future to support other authentication types such as 802.11x.
- 367f36ee1f5e272cfbca1ca35a73e42b: The leftmost 16 bytes of the SHA-256 (Secure Hash Algorithm 256) hash of simple device certificate in hexadecimal format.
- 34: A checksum value for verifying manual entry of the QR code data, in case the QR code scanning fails by the smartphone camera. This is the leftmost byte of the SHA-256 hash of the previous field.

The encoding of this message as the QR code is shown in Figure 28.

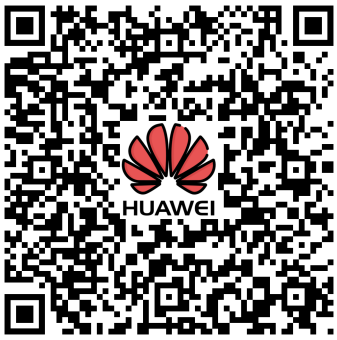

Figure 28: QR code OOB message example.

The checksum value was added to the message structure since the QR code scanning may fail, and the user may be required to type the encoded data in the QR code. To support manual typing, the values in the QR code are encoded as American Standard Code for Information Interchange (ASCII) characters. The QR code sticker has to include the same data also as text.

The message format does not use fixed-length fields. This provides flexibility to the message structure. For example, in the future, if the hash length or the WiFi PSK is found to be short, then the values can be extended.

During the bootstrap procedures, an attacker can sniff the 802.11 4-Way Handshake and then mount a Denial-of-Service (DoS) attack on the WiFi connection. Then, the attacker can crack the PSK with offline brute force attack. However, using a strong PSK (in the PoC prototype it is 128-bit) mitigates this attack.

4.4 Results

This thesis integrates the OOB channel communication and the LwM2M standard to securely manage the lifecycle of IoT devices. The designed system uses the OOB

channel to identify each IoT device uniquely. The OOB channel message contains information about the simple device credentials and in-band channel configuration. The simple device credentials may be a PSK or an RPK or an X.509 certificate. If the simple device uses PSK, the PSK is transferred over the OOB channel. If the credential is in the form of an RPK or X.509 certificate, the hash of the public key or certificate is transferred.

The security of the system depends on the following assumptions. First, it is assumed that the OOB channel message cannot be altered or spoofed. Depending on the simple device credentials, it might be vital that the OOB channel message is not eavesdropped. If the simple device uses the PSK authentication method, then the confidentiality of the OOB channel message is vital, because the PSK will be available as plain text in the OOB channel message. Once the OOB channel message is eavesdropped, then the communication between the simple device and the controller is compromised. Second, the hash algorithm that generates the digest value from the certificate and the other used protocols like DTLS is secure. Hence, creating another certificate with the same hash value is not feasible with the current computation power. Third, the controller is not the attacker.

5 Discussion and Analysis

5.1 One-way vs. Two-way OOB

In the designed system in this thesis, the controller device is not authenticated. The first controller which connects to the simple device takes control of it. The uses the "resurrecting duckling" security policy [33].

The resurrecting duckling security policy may lead impersonation attack for the controller. A malicious controller may try to mount an impersonation or MitM attack on the WiFi in-band channel between the bootstrap server and the simple device during bootstrap. Then, the malicious controller may try to take control of the simple device by accepting any certificate or RPK hash before the legitimate controller since the controller is not authenticated. However, if the simple device uses a WiFi PSK that is resistant to brute force attacks and the OOB channel message is not eavesdropped, the malicious controller cannot mount this attack.

In this thesis, a QR code is used as the OOB channel. Nowadays, most smartphones have a built-in camera. Therefore, the QR code scanning method can be supported by most smartphones. However, the QR code scanning transfers data only from the simple device to the controller; not the other way. Due to this limitation, implementing mutual authentication of the parties during the bootstrap is challenging. Note that mutual authentication could be done also in this case by using a symmetric key, which is transferred over OOB interface during the bootstrap. However, using symmetric key has its own weaknesses. For example, if an attacker manages to read the content of the QR code, then in the case of symmetric key, all security of the bootstrap is lost, while in the case of hash of RPK or certificate, the attacker still has to find a second-preimage of the hash.

In general, mutual authentication between the controller and the simple device can be done if both of them support a two-way OOB channel such as NFC. On the other hand, such OOB channels would require extra hardware in the simple device.

5.2 Denial of Service Attacks

In the PoC prototype, the Raspberry Pi is powered up with wall plug. However, in the case that the device is battery powered and accessible from the internet, an attacker may try to consume the battery power to cause denial of service.

Denial of service attack might be accomplished by sending a large number of data packets to the simple device. While the device tries to handle the excessive incoming data, the battery of the device will drain faster.

However, malicious incoming traffic does not present a great risk in a home network setting, since home networks are typically located behind a network address translator (NAT) and the devices in the local network are not accessible from outside.

5.3 Data Storage

The LwM2M Objects are stored as binary files without encryption in the Raspberry Pi file system. The LwM2M Security Object holds information about the client certificate, private key, and authentication method. Figure 29 shows the LwM2M Security Object in the file system. If somebody gains write access to the file system, they can easily modify the object data. For example, the *SECURITY_MODE* field, which is only one-byte length, determines the authentication method for the LwM2M Server connection. Changing this one byte will cause the client not to use encryption. However, the LwM2M Server should reject the authentication if the method does not match the authentication method in the corresponding client account. For example, the private key of the client is also stored as unencrypted in the object data. On the other hand, if someone gains read access to the memory, they can read the private key, then impersonate the simple device. These objects must be protected.

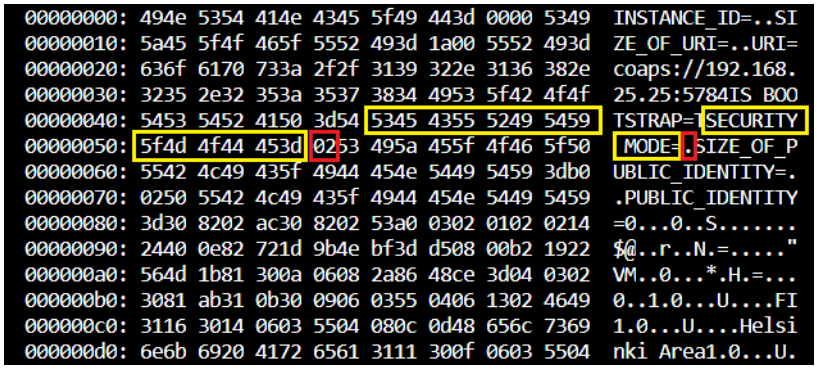

Figure 29: Security object in the file system.

Raspberry Pi is a single-board computer that does not have a hard disk. It stores its file system on an SD Card. Hence, the LwM2M Objects are stored on the Secure Digital (SD) Card in the experimental setup. An SD Card is a storage media with a NAND flash with an integrated controller. NAND flash memory can withstand 100,000 write cycles during the lifetime [34]. Therefore, the integrated controller in the SD Card tries to distribute the write counts over the whole memory to minimize wearing out of the flash memory. This protection mechanism has the notion of logical page numbers and physical page numbers. It acts as an intermediate block mapper between the logical blocks and physical blocks. Due to this mechanism, the real written blocks for an LwM2M Object cannot be determined by the operating system. While wiping an LwM2M Object from the file system, because of this intermediate controller in the SD Card, it cannot be guaranteed that the object data is wiped securely. Storing the objects in secure hardware storage or storing the objects in encrypted form and storing the key in secure hardware storage mitigates this data erasure problem.

In real-life implementation, the simple device should have a secure hardware storage module or a raw flash memory without the intermediate controller. This ensures that the stored data is wiped securely from the simple device.

It is important to check that the running firmware in the simple device is not modified throughout the lifecycle of the simple device. Not checking the running software allows an attacker to inject malware or entirely replace the firmware, leaving the device vulnerable. Anyone who has write access to the memory of the simple device can modify or corrupt the software. For example, an attacker could install his own malicious firmware to the simple device, and then he can sell the device as a second-hand device. The new owner cannot know whether the simple device runs genuine or malicious firmware. There is no built-in protection mechanism for this type of attack in the PoC prototype. However, this threat can be mitigated by using secure boot.

Secure boot is a mechanism that prevents the execution of unauthorized code when the device boots. This can be accomplished in several ways. One way is to allow only the execution of binaries which are signed by the device manufacturer. The other way is using trusted boot loaders and security microprocessors. By employing the secure boot, any code modification is detected and the device does not run the modified software.

5.4 Key Length

The PoC prototype uses 256-bit Elliptic Curve Cryptography (ECC) public key based on the NIST P-256 curve (also known as secp256r1 and prime256v1). The public key is used to authenticate the simple device to the controller. 256-bit ECC public key has the same level of security as a 128-bit symmetric key or 3072-bit RSA key, as Table 4 shows. According to NIST [35], a 128-bit key is considered to be secure until 2030. Also according to ECRYPT [36], this key length is considered to be secure until 2028.

Security strength (bits)	Key size (bits)	
	ECC	RSA/DSA/DH
80	160	1024
112	224	2048
128	256	3072
192	384	7680
256	512	15360

Table 4: Security and key length comparison of ECC and RSA [37].

5.5 Random Number Generation

As mentioned in Section 5.4, the prototype uses Elliptic Curve Cryptography. During the encrypted session establishment, the prototype uses Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) to exchange keys. The simple device uses Elliptic Curve Digital Signature Algorithm (ECDSA) to identify itself to the LwM2M Bootstrap Server by creating a valid signature over the messages.

In ECDSA, generating cryptographically secure random numbers is crucial. Using

faulty random number generators or using the same random number more than once in ECDSA may leak the device's private key. There is a real-world example of ECDSA private key leakage when it is deployed with poorly implemented random number generators. In Chaos Communication Congress in 2010, a hacker group called fail0verflow presented a way to reveal the ECDSA signing key from PlayStation 3 gaming consoles [38]. The root cause of the vulnerability was the faulty random number generator implementation.

The random number generation problem can be solved by using the deterministic ECDSA [39] or Edwards-Curve Digital Signature Algorithm (EdDSA) [40]. The deterministic ECDSA and EdDSA were introduced in RFC 6979 and RFC 8032, respectively. In deterministic ECDSA and EdDSA, the random value is derived from the private key and the messages [39, 40]. Thus, these signature algorithms are secure to use in systems that lack secure random generators.

5.6 Code Size

The Wakaama project was integrated with MbedTLS and WiringPi to create the PoC prototype. Table 5 shows the additions to the Wakaama base project and Table 6 shows the details of the integration. LwM2M WLAN Object and Light Control Object were added to the base project. The Wakaama project can store the LwM2M Objects only in RAM. To store objects persistently, the object storage functionality was added. The Wakaama project supports only the PSK authentication method. With MbedTLS integration, the project supports also certificate-based authentication. The most time-consuming part of the project was the MbedTLS integration part. MbedTLS is a flexible library; however, it does not automatically manage the handshake steps and connection state. It is necessary to manage the states by adding extra controls. Also, the Wakaama project uses internal timers to manage the registration and bootstrap states. MbedTLS has to cooperate with the internal timers. Due to these limitations, the implementation took a couple of months.

Components	**Lines of code**
Baseline	34900
Additions	3500
Total	38400

Table 5: Our additions to the Wakaama project (only C source files and scripts, rounded to the nearest hundred).

The executable code size of the Wakaama project without any encryption support is 124 KB. After integrating the project with MbedTLS, the executable code size increased to 304 KB. Table 7 shows the executable code size for the base project and the PoC prototype. Furthermore, all these executable code was compiled with GNU Compiler Collection (gcc) optimization flags enabled. Most of the increase is caused by the multiple cipher suites support and X.509 certificate support. The implementation

Components	Lines of code
Object storage	500
MbedTLS integration	1400
Light control object	400
WLAN management	100
WLAN object	500
Other	600

Table 6: The details of our additions to Wakaama project (only C source files and scripts, rounded to the nearest hundred).

supports six different cipher suites as Figure 30 shows. The executable code size can be decreased by limiting the number of the cipher suites. To calculate the executable code size properly, MbedTLS library is statically linked to the executable file [5]. The WiringPi library cannot statically be linked to the executable code. Therefore, it is used as a shared library.

Components	Code size (KB)
Wakaama project base	124
Wakaama + MbedTLS	304

Table 7: Executable code size of the PoC prototype.

```
static const int CIPHER_SUITES[] = {
    MBEDTLS_TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256,
    MBEDTLS_TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,
    MBEDTLS_TLS_ECDHE_ECDSA_WITH_AES_256_CCM_8,
    MBEDTLS_TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,
    MBEDTLS_TLS_ECDHE_ECDSA_WITH_AES_128_CCM,
    MBEDTLS_TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8,
    0
};
```

Figure 30: Supported cipher suites in the PoC prototype.

The executable code may also require an operating system depending on the deployed platform. For example, the binary code can be integrated with the Free Real-time Operating System (FreeRTOS). FreeRTOS requires around 9 KB ROM memory [6]. As a result, the binary code and the operating system sum up to 313 KB. The total code size cannot fit a Class 2 IoT device, which has 250 KB flash memory (see Table 2). However, we believe that the code can further be optimized to fit the code into a Class 2 IoT device.

[5] http://wiringpi.com/wiringpi-deprecated
[6] https://www.freertos.org

5.7 Hash Length

The OOB channel message contains the truncated hash of the client RPK or X.509 certificate. Two of the important properties of a cryptographic hash function are second-preimage resistance and collision resistance.

Second-preimage resistance means that the probability of finding another value which results in the same hash output is negligible. The collision resistance means that the probability of finding two distinct values which result in the same hash value is negligible [41].

Truncating the hash output decreases the collision resistance of the hash value. Therefore, the collision resistance of SHA-256 truncated to 128 bits is $128/2 = 64$ bits.

In the case that the manufacturer is the attacker, for 128-bit truncated output of the SHA-256 hash function, the collisions can be generated with a birthday attack at a computational cost that is similar to brute forcing a hash of half of the truncated hash length. On the other hand, if the manufacturer is the attacker, it can do more serious damage than creating collisions in certificate hashes by, e.g, device cloning, putting a backdoor in the simple device firmware and impersonation using the private key.

The following formula approximates the collision probability of the hash values:

$$p(n,k) \approx 1 - e^{-\frac{k(k-1)}{2n}} \quad [42].$$

In the formula, k refers to the number of generated hash values, n refers to the number of possible hash values, and $p(n,k)$ refers to the collision probability after k hash values have been generated. Table 8 shows $p(n,k)$ for various values of n and k.

	$p(n,k)$: probability of a random hash collision					
n: nr. of possible hash values	10^{-9}	10^{-6}	10^{-3}	0.01	0.25	0.5
2^{32}	3	93	$2.9\text{x}10^{3}$	$9.3\text{x}10^{3}$	$5\text{x}10^{4}$	$7.7\text{x}10^{4}$
2^{64}	190,000	6,100,000	$1.9\text{x}10^{8}$	$6.1\text{x}10^{8}$	$3.3\text{x}10^{9}$	$5.1\text{x}10^{9}$
2^{128}	$8.2\text{x}10^{14}$	$2.6\text{x}10^{16}$	$8.3\text{x}10^{17}$	$2.6\text{x}10^{18}$	$1.4\text{x}10^{19}$	$2.2\text{x}10^{19}$
2^{256}	$1.5\text{x}10^{34}$	$4.8\text{x}10^{35}$	$1.5\text{x}10^{37}$	$4.8\text{x}10^{37}$	$2.6\text{x}10^{38}$	$4.0\text{x}10^{38}$

Table 8: The work required to find a hash collision at a given probability for different hash sizes.

A man-in-the-middle attacker who is trying to intercept the in-band communication during the bootstrap has to find a certificate whose truncated hash value matches with the hash value in the OOB message. On the other hand, it does not help this attacker to find a collision: two certificates with identical hash value, which does not

match that in the OOB message. For that reason, SHA-256 truncated to 128 bits, which has 128 bits level of second-preimage resistance, should be sufficient for our scenario.

Another threat arises when, due to technical problem or malicious intent, the manufacturer produces certificates which are very likely to have the same hash value. In that case, the man-in-the-middle attack can be done, for example, as follows; An attacker obtains a set of devices from the manufacturer and extracts their private keys. Then, he can try to intercept the in-band bootstrap messages, and if he has a matching certificate and key pair, then he can succeed to become main-in-the middle.

5.8 Discussion

Security is one of the key issues in consumer IoT. In this thesis, a system has been designed and implemented based on the Open Mobile Alliance Lightweight Machine-to-Machine specification to securely bootstrap, manage, control, and use IoT devices.

A PoC prototype was implemented based on the designed system with two Raspberry Pies and one Android smartphone. The Android smartphone is the controller. One of the Raspberry Pies is the LwM2M Client and the other one is the gateway device which hosts the LwM2M Bootstrap Server and LwM2M Server. The controller scans the QR code on the IoT device and sends the decoded data to the gateway device to identify and authenticate the IoT device. The controller is not authenticated, and the client trusts the first controller that connects to it. The Wakaama LwM2M Client project was modified and integrated with MbedTLS. The client controls an LED light with LwM2M messages. The LwM2M Client is bootstrapped with the LwM2M Bootstrap Server in the gateway device, and then the client is registered to the LwM2M Server.

A registered LwM2M Server can register a new LwM2M Server, return the device to factory settings, and update firmware in the simple device. Therefore, the permissions for the LwM2M Server should be restricted if there is more than one LwM2M Server (see Section 2.1.2).

The Leshan LwM2M project was modified to support identification and authentication based on the OOB channel message by Amel Bourdoucen [11], which compared the hash in the QR code with the certificate hash that was received from the LwM2M Client during the DTLS handshake. It would be easier to implement a custom-designed protocol with the OOB channel message to demonstrate that the designed system works. However, the custom protocol would not be suitable for heterogeneous IoT networks since it would require more development time and integration time for each type of device. Using the LwM2M protocol saves design and implementation time since it is an open standard and open-source implementations are available. Also, the standard protocols are reviewed by more people; thus, they have more secure design compared to custom protocols. For different types of devices, the relevant LwM2M objects already exist in the OMA object registry. Currently, there

are around 300 LwM2M objects in the registry [7]. Therefore, it is easy to deploy the system to different IoT devices.

As a result, the designed system is practical and applicable to home IoT devices. Since it uses standard protocols, it requires less design and implementation time compared to custom ones.

5.9 Limitations

The Wakaama project has not been very active since February 2019 and there were more than 50 open issues in the project. Also, there was no documentation about the project. Therefore, the PoC implementation was challenging.

The PoC project was developed for Raspberry Pi. Raspberry Pi does not have a built-in screen or a powerful graphic processor. Therefore, it is not suitable to write code directly on a Raspberry Pi due to the limited computation power. It was necessary to build a development environment for Raspberry Pi. The file system of the Raspberry Pi was mounted as a network folder over Secure Shell (SSH) to the development computer. Then, the code was developed on a development laptop and synchronized over the SSH connection. The code was compiled and tested in the Raspberry Pi over SSH.

Initially, it was planned to port Leshan Server implementations to Android operating system (OS).

The PoC prototype was designed and implemented during the global COVID-19 pandemic, and there were governmental restrictions like movement restrictions and social distancing regulations. Not being able to present in the company made the testing and implementation more challenging. These limitations caused further challenges during the implementation.

5.10 Future Work

For future work, the server authentication and secure key storage functionalities can be added to the system. In the PoC prototype, static IP addresses were used. It can be improved with Multicast Domain Name System (mDNS). Also, the other system configuration mentioned in Section 4 can be implemented. The LwM2M Server and LwM2M Bootstrap Server could not be ported to Android due to several limitations [11]. The servers can be ported to Android.

[7] https://github.com/OpenMobileAlliance/lwm2m-registry

6 Conclusion

Device lifecycle management is a set of integrated steps that defines the entire lifecycle of a product including design, manufacturing, deploying, updating and disposing. The management of IoT devices during their lifecycle is one of the key issues in the IoT networks.

This thesis presents a system design that combines QR code scanning and the Lightweight machine-to-machine (LwM2M) protocol to securely manage the lifecycle of IoT devices starting from the manufacturing throughout the device lifetime. For the QR code scanning integration, the LwM2M specification has been modified as little as possible.. Additionally, the designed system defines the required procedures and steps to secure the lifecycle of IoT devices and user data inside them. The devices are paired and controlled with a smartphone. A proof-of-concept (PoC) prototype has been implemented with a Raspberry Pi, which acts as the IoT device, and a smartphone, which acts as the controller. The designed system and the PoC prototype have been analyzed in various aspects, and the main findings have been presented.

In the designed system, the user scans the QR code of the device with his/her smartphone's camera to start the initial setup of the IoT device. The security of the initial setup is based on the metadata and the information of the IoT device's credentials in the QR code, which may include, e.g., the pre-shared key of the device or the hash of the device's public key. After the initial setup, the IoT device joins to the home WiFi network, and it is controlled via the user's smartphone.

The designed system authenticates and uniquely identifies each IoT device with the information in the QR code. The communication channels between the IoT device and the controller are secured by Datagram Transport Layer Security (DTLS). The details and implementation of the controller part are the subjects of another master thesis that was done parallelly in the same project [11].

As a result, the designed system is practical and applicable to consumer IoT. The designed system is based on open standards and commonly used technologies. Therefore, it requires less design and implementation time compared to custom-made protocols.

References

[1] Kevin Ashton, *How to Fly a Horse: The Secret History of Creation, Invention and Discovery*. Doubleday, 2015, pp. 9–13.

[2] Ericsson. Internet Mobility Report, November 2019. https://www.ericsson.com/en/mobility-report/reports/november-2019. Accessed: 01.06.2020.

[3] Alma Oracevic, Selma Dilek, and Suat Özdemir, "Security in internet of things: A survey," in *2017 International Symposium on Networks, Computers and Communications (ISNCC)*. Marrakech, Morocco: IEEE, May 2017, pp. 1–6.

[4] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas, "DDoS in the IoT: Mirai and other botnets," *Computer*, vol. 50, no. 7, pp. 80–84, July 2017.

[5] F-Secure. Attack Landscape H1 2019.

[6] Carsten Bormann, Mehmet Ersue, and Ari Keränen, "Terminology for Constrained-Node Networks," Internet Requests for Comments, RFC Editor, RFC 7228, May 2014.

[7] Leila Fatmasari Rahman, Tanır Özçelebi, and Johan Lukkien, "Understanding IoT systems: A life cycle approach," *Procedia Computer Science*, vol. 130, pp. 1057–1062, 2018.

[8] Apple Inc., "HomeKit Accessory Protocol Specification (Non—Commercial Version) Release R1," Tech. Rep., 2017.

[9] Open Mobile Alliance, "Lightweight Machine to Machine Technical Specification: Core V1.1," 2018.

[10] Zach Shelby, Klaus Hartke, and Carsten Bormann, "The Constrained Application Protocol (CoAP)," Internet Requests for Comments, IETF, RFC 7252, June 2014.

[11] Amel Bourdoucen, "Securing Communication Channels in IoT using an Android Smart Phone," Master's thesis, Department of Future Technologies, University of Turku, Finland, June 2020.

[12] Matthias Kovatsch, Simon Duquennoy, and Adam Dunkels, "A Low-Power CoAP for Contiki," in *2011 IEEE Eighth International Conference on Mobile Ad-Hoc and Sensor Systems*, Valencia, Spain, November 2011, pp. 855–860.

[13] Shahid Raza, Hossein Shafagh, Kasun Hewage, René Hummen, and Thiemo Voigt, "Lithe: Lightweight secure coap for the internet of things," *IEEE Sensors Journal*, vol. 13, no. 10, pp. 3711–3720, August 2013.

[14] Christian Lerche, Laum Nico, Frank Golatowski, Dirk Timmermann, and Christoph Niedermeier, "Connecting the web with the web of things: lessons learned from implementing a CoAP-HTTP proxy," in *IEEE 9th International*

Conference on Mobile Ad-Hoc and Sensor Systems, Las Vegas, NV, USA, October 2012, pp. 1–8.

[15] Zach Shelby, Klaus Hartke, and Carsten Bormann, "The Constrained Application Protocol (CoAP)," Internet Requests for Comments, IETF, RFC 7252, June 2014.

[16] Telit IoT Platform. LWM2M Object model. https://docs.devicewise.com/Content/GettingStarted/LWM2M-Object-model.htm. Accessed: 11.06.2020.

[17] Hannes Tschofenig and Thomas Fossati, "Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things," Internet Requests for Comments, RFC Editor, RFC 7252, July 2016.

[18] Eric Rescorla and Nagendra Modadugu, "Datagram Transport Layer Security Version 1.2," Internet Requests for Comments, RFC Editor, RFC 6347, January 2012.

[19] Eric Rescorla and Nagendra Modadugu, "Datagram Transport Layer Security," Internet Requests for Comments, RFC Editor, RFC 4347, April 2006.

[20] Eric Rescorla, Nagendra Modadugu, and Hannes Tschofenig, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3, (draft)," Internet Requests for Comments, RFC Editor, RFC draft.

[21] Yassine Maleh, Abdellah Ezzati, and Mustapha Belaissaoui, "DoS Attacks Analysis and Improvement in DTLS Protocol for Internet of Things," in *Proceedings of the International Conference on Big Data and Advanced Wireless Technologies*. Blagoevgrad, Bulgaria: Association for Computing Machinery, November 2016.

[22] Maleh Yassine and Abdellah Ezzati, "Towards an Efficient Datagram Transport Layer Security for Constrained Applications in Internet of Things," *International Review on Computers and Software*, vol. 11, pp. 611–621, July 2016.

[23] Gack Davidson, *Apple Homekit: The Beginner's Guide*. CreateSpace Independent Publishing Platform, 2017.

[24] Dennis Giese, "Having fun with IoT: Reverse Engineering and Hacking of Xiaomi IoT Devices," in *DEFCON 26*, Las Vegas, Nevada, USA, August 2018, Accessed: 18.06.2020. [Online]. Available: https://media.defcon.org/DEF%20CON%2026/DEF%20CON%2026%20presentations/DEFCON-26-Dennis-Giese-Having-Fun-With-IOT-Updated.pdf

[25] Jose Garcia and Cihan Varol, "SmartThings Event Export using SmartApps," in *2019 7th International Symposium on Digital Forensics and Security (ISDFS)*, Barcelos, Portugal, June 2019, pp. 1–6.

[26] Kaniz Fatema Tuly, "A Survey on Novel Services in Smart Home (Optimized for Smart Electricity Grid)," Master's thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, Trondheim, 2016.

[27] SmartThings Inc. SmartThings Support, Z-Wave General Info. https://support.smartthings.com/hc/en-us/articles/204392790-Z-Wave-General-Info. Accessed: 17.06.2020.

[28] DENSO Corporation. (2011) QR Code Essentials. http://www.nacs.org/LinkClick.aspx?fileticket=D1FpVAvvJuo%3D&tabid=1426&mid=4802. Access: 05.06.2020.

[29] Katharina Krombholz, Peter Frühwirt, Peter Kieseberg, Ioannis Kapsalis, Markus Huber, and Edgar Weippl, "QR Code Security: A Survey of Attacks and Challenges for Usable Security," in *Human Aspects of Information Security, Privacy, and Trust*, Crete, Greece, June 2014, pp. 79–90.

[30] Huawei Security Protection Technology Lab, "Lifecyle Management of Simple Devices (unpublished Tech. Rep.)," , February 2020.

[31] Sampsa Latvala, Mohit Sethi, and Tuomas Aura, "Evaluation of Out-of-Band Channels for IoT Security," *SN Computer Science*, vol. 1, no. 1, September 2019.

[32] Tongguang Zhang, Shuai Zhao, Bingfei Ren, Shi Yulong, Bo Cheng, and Junliang Chen, "Performance enhancement of multipath tcp in mobile ad hoc networks," in *IEEE 25th International Conference on Network Protocols (ICNP)*.

[33] Frank Stajano and Ross Anderson, "The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks," in *Security Protocols, 7th International Workshop*. Cambridge, UK: Springer Berlin Heidelberg, April 1999, pp. 172–182.

[34] Bhupendra Singh, Ravi Saharan, Gaurav Somani, and Gaurav Gupta, "Secure File Deletion For Solid State Drives," in *Advances in Digital Forensics XII: 12th IFIP WG 11.9 International Conference, New Delhi, January 4-6, 2016, Revised Selected Papers*, ser. IFIP Advances in Information and Communication Technology, vol. 484. Springer International Publishing, 2016.

[35] Elaine Barker, "NIST Special Publication 800-57 Part 1 Revision 5, Recommendation for Key Management: Part 1 – General," Tech. Rep., May 2020, https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf.

[36] Nigel Smart, ECRYPT – Coordination & Support Action, "D5.4 Algorithms, Key Size and Protocols Report (2018)," Tech. Rep., February 2018, https://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf.

[37] Nurzhan Zhumabekuly Aitzhan and Davor Svetinovic, "Security and Privacy in Decentralized Energy Trading Through Multi-Signatures, Blockchain and Anonymous Messaging Streams," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 5, pp. 840–852.

[38] Yuval Yarom and Naomi Benger, "Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack," *IACR Cryptology ePrint Archive*, vol. 2014, February 2014.

[39] Thomas Pornin, "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)," Internet Requests for Comments, RFC Editor, RFC 6979, August 2013.

[40] Simon Josefsson and Ilari Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)," Internet Requests for Comments, RFC Editor, RFC 8032, January 2017.

[41] Mihir Bellare and Phillip Rogaway, "Collision-Resistant hashing: Towards making UOWHFs practical," in *Advances in Cryptology — CRYPTO '97*. Santa Barbara, CA, USA: Springer Berlin Heidelberg, August 1997, pp. 470–484.

[42] Ganesh Gupta, "What is Birthday attack??" February 2015. [Online]. Available: https://dx.doi.org/10.13140/2.1.4915.7443